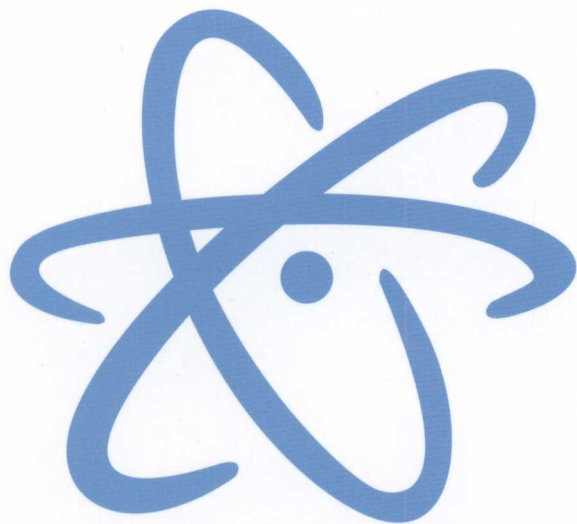


版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

React Native 移动开发实战

—— 向治洪 著 ——



**React Native 框架超全面解析！
手把手教你打造高品质移动用户体验！**



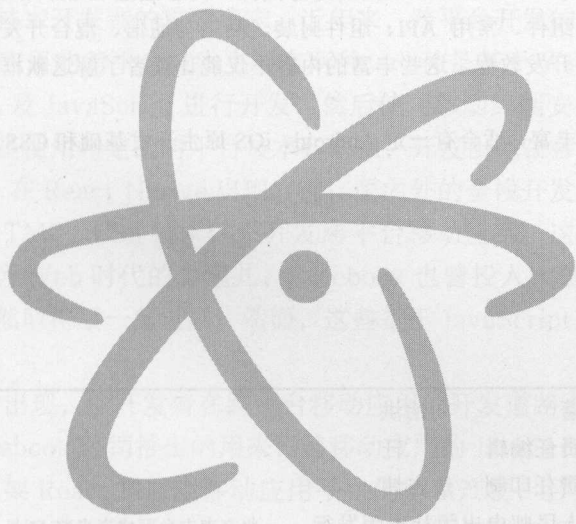
中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

React Native 移动开发实战

向治洪 著



人民邮电出版社

北京

图书在版编目 (CIP) 数据

React Native移动开发实战 / 向治洪著. — 北京 :
人民邮电出版社, 2018.1
ISBN 978-7-115-47096-6

I. ①R… II. ①向… III. ①移动终端—应用程序—
程序设计 IV. ①TN929.53

中国版本图书馆CIP数据核字(2017)第276301号

内 容 提 要

本书全面详尽地介绍了 React Native 框架的方方面面, 内容涵盖 React Native 基础知识、环境搭建与调试、开发基础、常用组件、常用 API、组件封装、网络与通信、混合开发、热更新与打包部署, 以及两个实际案例的完整开发教程。这些丰富的内容不仅能让读者了解这款框架中涉及的各类概念, 还能指导读者开发实践。

本书语言简洁, 内容丰富, 适合有一定 Android、iOS 原生开发基础和 CSS 基础的移动开发工程师学习。

-
- ◆ 著 向治洪
责任编辑 赵 轩
责任印制 焦志炜
 - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京市艺辉印刷有限公司印刷
 - ◆ 开本: 800×1000 1/16
印张: 21
字数: 496 千字 2018 年 1 月第 1 版
印数: 1—3 000 册 2018 年 1 月北京第 1 次印刷
-

定价: 69.00 元

读者服务热线: (010)81055410 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广登字 20170147 号

前言

PREFACE

近年来，随着移动互联网产业的持续快速发展，以及智能手机、智能电视等智能终端设备的普及，移动互联网应用获得了爆炸式增长。面对未来广阔的市场，运营商、互联网企业、设备生产商等产业巨头纷纷构建了移动互联网生态链，其中苹果公司和 Google 创造的移动互联网应用商业模式，激发了广大开发者开发移动互联网应用的热情。

众所周知，原生（Native）应用因其性能优秀、体验较好而获得了广大用户和开发者的欢迎。然而，原生应用开发周期长、支持设备有限等问题也困扰着开发者和商户，因而，跨平台移动应用开发成为技术开发者的重要追求。近年来，跨平台开发的呼声越来越高，已成为一种趋势。目前，移动应用的跨平台技术主要有两种。一种是基于 Web 的移动开发技术，只需要使用标准的 HTML 及 JavaScript 进行开发，然后使用移动终端安装的浏览器，即可实现应用的跨平台；另一种是使用特定的跨平台技术和框架，开发出能在各种主流移动操作系统上运行的 APP 应用程序。在 React Native 出现之前，国内外的全栈开发社区都在坚持不懈地寻求使用 JavaScript 和 HTML、CSS 技术体系开发跨平台移动应用，这些技术被统称为 H5 技术（HTML5 技术）。作为 Web 时代的弄潮儿，Facebook 也曾投入大量的人力物力，在移动 H5 技术上攻坚克难，虽然取得了一些进展，然而，这些基于 JavaScript 体系开发的移动应用始终达不到理想的效果。

React Native 的出现，使开发者在跨平台移动应用的开发道路上向前迈了一大步。React Native 是一款由 Facebook 公司推出的用来构建移动应用的 JavaScript 框架，是 Facebook 早先开源的界面渲染框架 React 在原生移动应用平台的衍生产物，目前支持 iOS 和 Android 两大平台。React Native 倡导的“Learn once, write anywhere”（仅需学习一次，编写任何平台）也赢得了广大开发人员的青睐。虽然，新框架的引入不可避免地增加了学习成本，但是，相对于其他的跨平台技术而言，React Native 的学习成本还是比较低的。截至 2017 年 7 月，在 GitHub 上 React Native 已获得了 52000 多个 star，成为时下最受欢迎的跨平台移动应用开发框架之一。

在技术实现上，React Native 抛弃了传统的浏览器加载思路，转而采用曲线调用原生 API 的思路来渲染界面，从而获得了媲美原生应用的体验。React Native 具体实现思路如下：应用启动后会从服务器下载最新的 JSBundle 文件，然后通过本地的 JavascriptCore 引擎对 JS（Javascript 缩写）文件进行解析，并利用 Bridge 映射到对应的原生 API 上，进而调用原生方法和 UI 组件来渲染界面。在语法上，React Native 使用 JSX 来替代常规的 JavaScript，这

是一种很像 XML 的 JavaScript 语法扩展。因此,熟悉 JavaScript 类库的 Web 开发者可以使用 React Native 轻松地开发出移动应用。由于使用 JSX 编写的大部分代码可以实现平台间共享,因此,采用 React Native 开发可以大幅减少跨平台移动应用开发的工作量。同时,React Native 框架采用模块化结构,使应用版本的更新迭代也异常简单。当然,React Native 也不是完美无缺的,但瑕不掩瑜,随着它的日趋成熟,React Native 势必会成为跨平台移动应用开发的主流技术。

本书适当地介绍了一些原理性的概念,但并不深究,同时本书提供的不少案例,也将带领你快速地进入 React Native 的世界。雄关漫道真如铁,而今迈步从头越。相信通过学习本书,你一定会有所收获。

作者

目录

CONTENTS

第1章 React Native入门

1.1 React Native基本知识	1
1.1.1 React简介	1
1.1.2 React Native简介	4
1.1.3 React Native工作原理	5
1.2 React Native与其他跨平台技术的 对比优势	6
1.2.1 Web流	7
1.2.2 代码转换流	7
1.2.3 编译流	8
1.2.4 虚拟机流	10
1.3 小结	11

第2章 React Native环境搭建与调试

2.1 React Native环境搭建	12
2.1.1 Mac环境下搭建React Native	12
2.1.2 React Native开发IDE	15
2.1.3 创建React Native项目	16
2.1.4 运行React Native项目	17
2.1.5 iOS环境	18
2.1.6 Android环境	19
2.1.7 Windows环境下搭建React Native	22
2.2 React Native 项目结构剖析	22
2.2.1 React Native文件结构	22
2.2.2 iOS文件结构及代码分析	23
2.2.3 Android文件结构及代码分析	24
2.3 React Native开发IDE介绍	26
2.3.1 Atom+Nuclide	26
2.3.2 WebStorm	29
2.4 React Native调试技巧	30
2.4.1 JavaScript调试技巧	30
2.4.2 React Native调试	33
2.5 React Native代码测试	36
2.5.1 使用Flow进行类型检查	36
2.5.2 Jest单元测试	37
2.5.3 集成测试	37

2.6 小结	38
--------------	----

第3章 React Native开发基础

3.1 FlexBox布局	39
3.1.1 FlexBox简介	39
3.1.2 FlexBox布局模型	40
3.1.3 FlexBox布局属性	41
3.1.4 FlexBox伸缩项目属性	45
3.1.5 FlexBox在React Native中的应用	47
3.1.6 FlexBox综合实例	48
3.2 ES6语法基础	50
3.2.1 组件的导入与导出	51
3.2.2 类	52
3.2.3 状态变量	53
3.2.4 回调函数	54
3.2.5 参数	55
3.2.6 箭头操作符	57
3.2.7 Symbol	57
3.2.8 解构	58
3.3 React JSX	60
3.3.1 JSX入门	60
3.3.2 JSX语法	61
3.4 样式	64
3.4.1 申明与操作样式	64
3.4.2 样式分类	64
3.4.3 样式使用	66
3.4.4 样式传递	67
3.5 手势与触摸事件	68
3.5.1 触摸事件	68
3.5.2 手势系统响应	70
3.5.3 辅助功能	74
3.6 小结	77

第4章 常用组件介绍

4.1 HTML元素与原生组件	78
4.1.1 文本组件	79
4.1.2 图片组件	80

4.1.3 TextInput组件	82	6.4 小结	192
4.1.4 ScrollView组件	87	第7章 网络与通信	
4.2 结构化组件	92	7.1 通信机制	193
4.2.1 View组件	92	7.1.1 React Native与Android通信	194
4.2.2 ListView组件	94	7.1.2 React Native与iOS通信	208
4.2.3 Navigator组件	101	7.2 Promise 机制	210
4.2.4 WebView组件	106	7.2.1 Promise 简介	210
4.3 平台特定组件	109	7.2.2 Promises基本用法	213
4.3.1 TabBarIOS和TabBarIOS.Item 组件	109	7.2.3 在React Native中使用AJAX技术	215
4.3.2 ToolbarAndroid组件	113	7.3 网络请求	216
4.3.3 SegmentedControlIOS组件	115	7.3.1 XMLHttpRequest请求	216
4.3.4 ViewPagerAndroid组件	117	7.3.2 fetch请求	218
4.4 Touchable系列组件	119	7.4 小结	223
4.4.1 TouchableWithoutFeedback	120	第8章 混合开发高级篇	
4.4.2 TouchableHighlight	120	8.1 React Native调用iOS原生组件	224
4.4.3 TouchableOpacity	122	8.1.1 React Native链接原生库	225
4.4.4 TouchableNativeFeedback	122	8.1.2 React Native调用Objective-C创建的 原生组件	227
4.5 小结	123	8.2 React Native调用Android原生组件	233
第5章 常用API介绍		8.2.1 编写原生UI组件	233
5.1 AppRegistry	124	8.2.2 编写JavaScript端实现	236
5.2 StyleSheet	126	8.3 小结	238
5.3 AppState	128	第9章 热更新与打包部署	
5.4 AsyncStorage	129	9.1 iOS应用打包	239
5.5 PixelRatio	132	9.1.1 iOS应用配置	240
5.6 Animated	133	9.1.2 打包离线Bundle	242
5.7 Geolocation	142	9.1.3 设置发布Scheme	243
5.8 NetInfo	144	9.1.4 发布应用	243
5.8.1 获取网络状态	144	9.2 Android应用打包	244
5.8.2 网络状态监听	145	9.2.1 打包离线Bundle	244
5.8.3 判断网络是否连接	146	9.2.2 生成签名密钥	245
5.9 小结	146	9.2.3 生成签名APK	246
第6章 组件封装		9.3 热更新	248
6.1 组件的生命周期	147	9.3.1 热更新原理	249
6.2 第三方库	150	9.3.2 热更新配置	249
6.2.1 react-navigation	150	9.3.3 登录与创建应用	252
6.2.2 react-native-tab-navigator	153	9.3.4 添加热更新功能	253
6.2.3 react-native-scrollable-tab-view	157	9.3.5 发布热更新版本	256
6.2.4 react-native-image-picker	161	9.4 小结	257
6.2.5 Mobx	166	第10章 基于LBS的天气预报应用开发	
6.2.6 react-native-art	172	10.1 需求分析与确定	258
6.3 自定义组件	177	10.1.1 需求分析	258
6.3.1 组件的导出导入	177	10.1.2 需求确定	260
6.3.2 TabbarView封装	178	10.1.3 整体功能分析	260
6.3.3 九宫格布局封装	181		
6.3.4 下拉刷新组件封装	185		

10.1.4 技术与架构分析.....	261	11.2.1 模块划分.....	291
10.2 项目设计.....	261	11.2.2 添加第三方库.....	292
10.3 程序入口与工具模块.....	263	11.3 项目搭建与工具模块开发.....	293
10.3.1 程序入口.....	263	11.3.1 程序入口.....	293
10.3.2 数据模型定义与数据解析.....	266	11.3.2 搭建主框架.....	294
10.3.3 数据存储.....	271	11.3.3 导航栏封装.....	298
10.3.4 工具类.....	273	11.3.4 WebView封装.....	303
10.4 模块开发.....	275	11.3.5 字体样式工具类.....	306
10.4.1 组件封装.....	276	11.4 功能开发.....	307
10.4.2 天气预报页面开发.....	276	11.4.1 分类导航入口开发.....	307
10.4.3 Navigation导航.....	285	11.4.2 专题活动开发.....	309
10.5 运行结果.....	286	11.4.3 商品列表开发.....	311
第11章 O2O移动团购应用		11.4.4 详情页面开发.....	313
11.1 需求分析.....	288	11.4.5 Modal分享弹窗开发.....	318
11.1.1 需求分析.....	288	11.5 完成开发.....	322
11.1.2 功能分析.....	289	11.5.1 添加闪屏页.....	322
11.2 应用设计.....	291	11.5.2 修改应用图标和名称.....	324
		11.6 小结.....	325

React Native入门

1.1 React Native基本知识

React Native 基于 React 框架而设计，因此，了解 React 有助于我们更好地理解 React Native。

1.1.1 React简介

React 是由 Facebook 推出的前端开发框架，其本身作为 MVC 模式中的 View 层来构建 UI，也可以以插件的形式应用到 Web 应用程序的非 UI 部分构建中，轻松实现与其他 JS 框架的整合。同时，React 通过对虚拟 DOM 的操作来控制真实的 DOM，从而得到页面的局部更新，提高了 GPU 渲染的性能，而 React 提出的模块化开发思路也大大提高了代码的可维护性。

React 的官方地址是 <https://github.com/facebook/react>。截至 2017 年 4 月，React 获得了超过 62K 的 star 和 11K 的 fork，这说明 React 得到了技术人员的普遍支持，正是由于这些原因，React.js 和 Vue.js、Angular.js 成为了当今最流行的三大前端框架。

讲到 React，就不得不提到组件（Component）的概念，它是 React 最基础的部分，其功能相当于 AngularJS 里面的 Directive，或是其他 JS 框架里面的 Widgets 或 Modules。Component 可以认为是由 HTML、CSS、JavaScript 和一些内部数据组合而成的模块，当然 Component 也可以由很多其他的 Component 组建而成。不同的 Component 既可以用纯 JavaScript 定义，也可以用特有的 JavaScript 语法 JSX 创建而成。采用 React 进行项目开发，能够获得以下优势。

虚拟DOM (Virtual DOM)

传统的 Web 应用开发，一般都是通过直接操作真实 DOM 来进行更新操作的，如图 1-1 所示，但对 DOM 进行操作通常是比较昂贵的。而 React 为了尽可能减少对真实 DOM 的操作，采用了一种强大的方式来更新 DOM，代替直接的 DOM 操作，这就是 Virtual DOM，一个轻量级的虚拟 DOM。

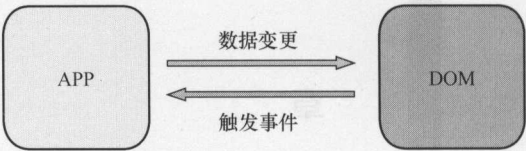


图1-1 传统的Web应用结构图

虚拟 DOM 其实是 React 抽象出一个对象，通过这个 Virtual DOM 可以更新真实的 DOM，由这个 Virtual DOM 管理真实 DOM 的更新，如图 1-2 所示。简单来说，React 在每次需要渲染时，会先比较当前 DOM 内容和待渲染内容的差异，然后再决定如何最优地更新 DOM，这个过程被称为 reconciliation。



图1-2 React Web应用结构图

除了性能方面的考虑，React 引入虚拟 DOM 更重要的意义在于提供了一种新的开发方式来开发服务端应用、Web 应用和手机端应用，如图 1-3 所示。

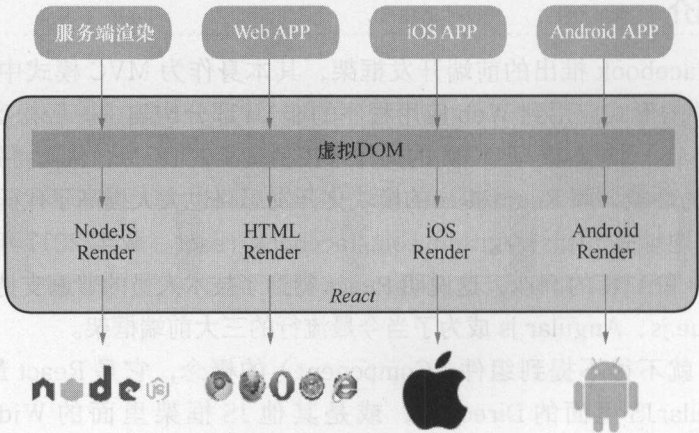


图1-3 虚拟DOM结构图

Components组件

虚拟 DOM (virtual-dom) 不仅带来了简单的 UI 开发逻辑，同时也带来了组件化开发的

思想。所谓组件，即自己封装的具有独立功能的 UI 部件。React 推荐以组件的方式去构成视图，并建议将功能相对独立的模块抽象为组件。例如，Facebook 的 `instagram.com` 网站都采用 React 来开发，整个页面就是一个大的组件。

对于 React 而言，界面被分成不同的组件，每个组件都相对独立。在 React 开发中，整个界面可以看成是由大小组件构成，每个组件实现自己的逻辑部分即可，彼此独立，如图 1-4 所示。

采用组件化开发，往往具有以下特点：

- 可组合（Composable）：一个组件易于和其他组件一起使用，或者嵌套在另一个组件内部。如果在一个组件内部创建了另一个组件，那么父组件拥有它创建的子组件，通过这个特性，一个复杂的 UI 可以拆分成多个简单的 UI 组件。
- 可重用（Reusable）：每个组件都可以独立出来，被使用在其他相似的 UI 场景中。
- 可维护（Maintainable）：每个小的组件仅仅包含自身的逻辑，更容易被理解和维护。
- 可测试（Testable）：每个组件都是独立的，那么对于各个组件分开测试显然要比整个界面容易得多。

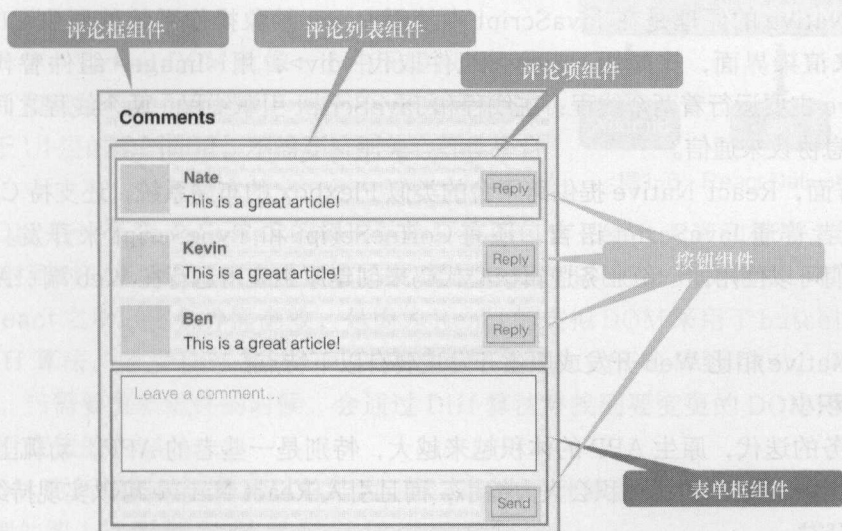


图 1-4 Components 组件示意图

数据流（Data Flow）

React 采用单向的数据流，即从父节点到子节点的传递，因此更加灵活便捷，也提高了代码的可控性。React 单向数据流可以总结为以下流程：Action → Dispatcher → Store → View，如图 1-5 所示。

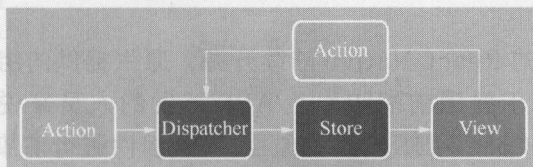


图 1-5 React 单向数据流流程图

JSX语法

JSX 是 React 的核心组成部分，React 使用 JSX 来替代常规的 JavaScript。它使用 XML 标记的方式去直接声明界面，目的是通过各种不同的编译器将这些标记编译成标准的 JS 语言。使用 JSX 语法后，可以让组件的结构和组件之间的关系看上去更加清晰并且执行效率更高。

1.1.2 React Native简介

React Native（简称 RN）是 Facebook 于 2015 年 4 月开源的跨平台移动应用开发框架，是 Facebook 早先开源的 UI 框架 React 在原生移动应用平台的衍生产物，目前支持 iOS 和 Android 两大平台。

React Native 可以基于目前大热的开源 JavaScript 库 ReactJS 来开发 iOS 和 Android 移动应用，因为往往只需要开发一套代码就可以满足 iOS 和 Android，正如 Facebook 说的“Learn once, write anywhere”（仅需学习一次，编写任何平台），由于基于 Web 技术，React Native 开发起来可以像在浏览器那样随改即所见。

React Native 的原理是在 JavaScript 中使用 React 抽象操作系统的原生 UI 组件，代替 DOM 元素来渲染界面，比如用 `<View>` 组件取代 `<div>`，用 `<Image>` 组件替代 `` 等。React Native 主要运行着两个线程：主线程和 JavaScript 引擎线程，两个线程之间通过批量化的 async 消息协议来通信。

在 UI 方面，React Native 提供跨平台的类似 Flexbox 的布局系统，还支持 CSS 子集。可以用 JSX 或者普通 JavaScript 语言，还有 CoffeeScript 和 TypeScript 来开发。运用 React Native，我们可以使用同一份业务逻辑核心代码来创建原生应用运行在 Web 端、Android 端和 iOS 端。

React Native 相比 Web 开发或原生开发主要有以下特点：

APP占用体积小

随着业务的迭代，原生 APP 的体积越来越大，特别是一些老的 APP，动辄上百兆，而采用 React Native 之后，占用体积会大大缩小，而且引入 React Native 可以实现持续开发。

实现跨平台开发

基于 Web 技术（HTML5/JavaScript）构建的移动应用速度快，开发周期短，但是体验较差，响应不及时；而使用原生开发，周期长，项目风险不可控。如何提高开发效率，节约人力成本。成为各大公司考虑的问题，而 React Native 的出现解决了上面的问题，只需要开发一套代码，便可以同时部署到 Android 和 iOS 两个移动平台上。

相对成熟的技术

随着 Android/iOS 的 React Native 陆续开源，原生提供的组件和 API 相对丰富，且实现技术基本一致，对于熟悉前端和原生 APP 开发的人员来说很容易上手。而 React Native 通过 JavaScriptCore 将 JS 转换为原生 APP 组件进行渲染，其用户体验也可媲美原生 APP。

支持动态更新

在原生 APP 开发中, Android 平台可以通过插件化实现热更新。在 iOS 平台上, 热更新策略是严令禁止的 (如 JSPatch/wax/rollout 等技术), 而采用 React Native 技术完全可以满足要求, 而又不触碰苹果的底线。

1.1.3 React Native工作原理

使用 JavaScript 开发移动 APP 的想法来源于最近几年市场对于移动应用需求的增长, 为了快速开发一款可以使用, 而体验又不是那么糟糕的 APP, 很多公司投入大量人力开发跨平台应用。而在这些公司当中, Facebook 无疑是做得最好、最成功的。

为了更好地理解 React Native, 我们需要对 React Native 的工作原理和整体架构有一个了解, 如图 1-6 所示。

如图, React Native 框架分为 3 层, 分别为: JSX 环境层、虚拟 DOM 层、具体的平台层。这里面最重要的就是虚拟 DOM 层。

在 React 中, Virtual DOM 就像一个中间虚拟层, 位于 JavaScript 和实际渲染页面之间。对于 JS 开发者来说, 只需要专注于 UI 层的绘制即可, 不需要特别关心具体平台的实现。

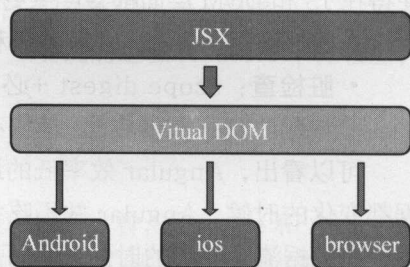


图1-6 React Native整体框架图

虚拟 DOM 是一个 JavaScript 的树形结构, 主要包含 React 元素和模块。组件的 DOM 结构就是映射到对应的虚拟 DOM 上, React 通过渲染虚拟 DOM 到浏览器, 使得用户界面得以显示。React 之所以更新界面高效, 是因为 React 的虚拟 DOM 采用了 batching (批处理) 和高效的 Diff 算法, 采用 Diff 算法, 可以将时间复杂度从 $O(n^3)$ 降到 $O(n)$, 从而提高界面构建的性能。当需要更新组件的时候, 会通过 Diff 算法寻找到要变更的 DOM 节点, 然后通知浏览器更新变化的内容。

虚拟 DOM 更新视图的过程可以总结为: 状态变化 → 计算差异 (Diff 算法) → 界面渲染。其渲染的原理如图 1-7 所示。

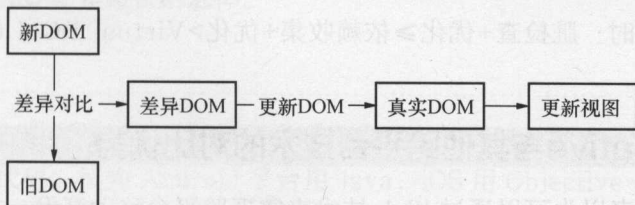


图1-7 虚拟DOM更新原理图

在界面渲染过程中, React Native 针对不同的平台调用原生 API 去渲染界面, 例如 iOS 平台调用其原生的 API 去渲染 iOS 界面, Android 平台调用其原生 API 去渲染 Android 界面, 而

不是直接渲染到浏览器的 DOM 上，这使得 React Native 不同于基于 Web 视图的跨平台应用开发方案，因而，采用 React Native 开发的 APP，体验更加接近原生 APP。

虚拟DOM和MVVM的对比

虚拟 DOM 只是 MVVM 框架的一种实现方案，二者没有好坏之分。在流行的前端框架中，除了 React 采用虚拟 DOM 之外，其他 MVVM 系框架，如 Angular、Vue、Avalon，采用的都是数据绑定。

何为数据绑定？简单来说，就是通过观察 Directive/Binding 对象数据变化，并保留对实际 DOM 元素的引用，当有数据变化时进行对应的操作。React 检查是 DOM 结构层面的，而 MVVM 的检查则是数据层面的。MVVM 的性能检测也根据检测层面的不同而有所不同：Angular 的脏检查使得任何变动都会产生固定的更新的代价；而 Vue/Avalon 采用的依赖收集，使得在 JS 和 DOM 层面都会产生更新。

上面提到两个概念——脏检查和依赖收集。

- 脏检查：scope digest + 必要DOM更新
- 依赖收集：重新收集依赖+必要DOM更新

可以看出，Angular 效率低的地方在于任何小变动都会引起界面的重绘，但是，当所有数据都变化的时候，Angular 并不吃亏。依赖收集在初始化和数据变化的时候都需要重新收集依赖，在数据流比较小的时候几乎可以忽略，但在数据量比较大的时候就会产生一定的消耗。相比之下，React 的变动检查则是 DOM 结构层面的，即使是全新的数据，只要渲染结果没有变化，也不需要重新绘制。

Angular 和 Vue 都提供了重绘的优化机制，即有效地复用实例和 DOM 元素。在优化的版本中，Angular 和 Vue 采用了 track by \$index 技术后比 React 的效率更高。

所以在框架选择和技术性能分析的时候，要分清楚初始渲染、小量数据更新、大量数据更新这些不同的场合，以及 DOM、脏检查 MVVM、数据收集 MVVM 在不同场合各自的表现和优缺点，具体表现和区别如下。

- 初始渲染阶段：Virtual DOM > 脏检查 ≥ 依赖收集
- 小量数据更新时：依赖收集 > Virtual DOM + 优化 > 脏检查（无法优化）> Virtual DOM 无优化
- 大量数据更新时：脏检查 + 优化 ≥ 依赖收集 + 优化 > Virtual DOM（无优化）> MVVM 无优化

1.2 React Native与其他跨平台技术的对比优势

曾经大部分开发者以为可以通过 Web 技术来实现跨平台移动开发，却因为性能限制或其他问题而放弃，最终，不得不针对多个平台开发多个版本，这违背了跨平台开发的初衷。而 React Native 的出现让跨平台移动端开发再次回到人们的视野中，而它提倡的“Learn once, write anywhere”也赢得了广大开发人员的青睐。相比传统的 H5 技术，React Native 获得了

更加接近原生应用的体验。

为了方便理解，笔者将跨平台技术分为四大流派。

- Web流：也被称为Hybrid技术，它基于Web相关技术来实现界面及功能。
- 代码转换流：将某个语言转成Objective-C、Java或C#，然后使用不同平台下的官方工具来开发。
- 编译流：将某个语言编译为二进制文件，生成动态库或打包成apk/ipa/xap文件。
- 虚拟机流：通过将某个语言的虚拟机移植到不同的平台上来运行。

1.2.1 Web流

Web流，如大家熟知的 PhoneGap/Cordova 等技术，它将原生的接口封装后暴露给 JavaScript，然后通过系统自带的 WebView 运行，也可以使自己内嵌 Chrome 内核。

Web流缺点是性能差、渲染速度慢。说它 Web 性能差，主要说的是在 Android 下比较差，在 iOS 下已经很流畅了。

性能差的主要原因是，在 Android 和 iOS 的早期设备中，由于没有实现 GPU 加速，所以会造成每次重绘界面的卡顿。

而造成渲染慢的第二个原因是：CSS 过于复杂。因为从实现原理上看，Chrome 和 Android View 并没有本质上的差别，但过于复杂的 CSS 会加重 GPU 的负担。那是不是可以通过简化 CSS 来解决呢？实际上还真有人进行了这种尝试，比如著名的 Famo.us，其最大的特色就是不使用 CSS，只能使用固定的几种布局方法，完全依靠 JavaScript 来写界面，它能有效避免低效的 CSS 代码，从而提升机器性能。

造成绘制缓慢的第三个原因是，业务需求的复杂，比如超长的 ListView 商品展示。因为 DOM 是一个很上层的 API，使得 JavaScript 无法做到像 Native 那样细粒度地控制内存及线程，所以难以进行优化，特别是在硬件较差的机器上。

上面三个问题现在都不好解决。其实除了性能之外，Web流更严重的问题是功能缺失。比如 iOS 8 就新增 4000 多个 API，而 Web 标准需要漫长的编写和评审过程，而等到 Web 审核通过，即便是 Cordova 这样的优秀的框架，或者自己封装也是忙不过来的。所以为了更好地使用原生系统新功能，Native 是最快的选择。

1.2.2 代码转换流

不同平台下的官方语言不一样，并且平台对官方语言的支持最好，这就导致对于同样的逻辑，我们需要写多套代码。比如 Android 平台用 Java，iOS 用 Objective-C 或者 Swift。于是就有人想到了通过代码转换的方式来减少重复的工作量，这就是代码转换流。

这种方式虽然听起来不是很靠谱，但它的成本和风险都是最小的，因为代码转换后就可以用官方提供的各种工具了，和普通开发区别不大，而且转换后，利用原生的优势，可以减少兼容性问题。

目前存在以下几种代码转换方式。

将Java转成Objective-C

2objc 是一款能将 Java 代码转成 Objective-C 的工具，据说 Google 内部就是使用它来降低跨平台开发成本的，比如 Google Inbox 项目就号称通过它共用了 70% 的代码，效果很显著。有了 2objc，我们就可以先开发 Android 版本，然后再开发 iOS 版本。

将Objective-C转成Java

MyAPPConverter 是一款将 Objective-C 代码转换成 Java 代码的工具，比起前面的 2objc，MyAPPConverter 还打算将 UI 部分也包含进来，从它已转换的列表中可以看到还有 UIKit、CoreGraphics 等组件，使得有些应用可以不改代码就能转换成功。

XMLVM

除了上面提到的源码到源码的转换，在代码转换流中，还有 XMLVM 这种与众不同的转换方式，它首先将字节码转成一种基于 XML 的中间格式，然后再通过 XSL 来生成不同语言，目前支持生成 C、Objective-C、JavaScript、C#、Python 和 Java。

虽然基于中间字节码可以支持多语言，但是这种方式也有一些问题，例如生成代码不可读，因为很多语言中的语法会在字节码中被抹掉，并且是不可逆的，所以不利于代码的调试和发现问题。

综上所述，虽然代码转换这种方式风险小，但对于很多小 APP 来说其实共享不了多少代码，因为这类应用大多数围绕业务来开发的，大部分代码都和业务逻辑耦合，所以公共部分不多，其意义不大。

1.2.3 编译流

编译流比代码转换流的代码转换更进一步，它直接将某个语言编译为普通平台下能够识别的二进制文件。采用这种方式主要有以下特点。

优点

- 可以重用一些实现很复杂的代码（比如之前用 C++ 实现的游戏引擎，重写一遍的成本太高）。
- 编译后的代码反编译困难，安全性更好。

缺点

- 转换过于复杂，并且后期定位和修改成本会很高。
- 编译后体积太大，尤其是支持 ARMv8 和 x86 等 CPU 架构的时候。

常用的编译流方案如下所示。

C++方案

因为目前 Android、iOS 和 Windows Phone 都提供了对 C++ 开发的支持。特别是 C++ 在实现非界面部分，性能是非常高的。而如果使用 C++ 实现非界面部分，还是比较有挑战的。这主要是因为 Android 程序的界面绝大部分是 Java 编写的，而在 iOS 和 Windows Phone 平台下

可以分别使用 C++ 的超集 Objective-C 和 C++/C# 来开发。要解决使用 C++ 开发 Android 应用程序界面的问题，目前主要有两种方案。

- 通过 JNI 调用系统提供的 Java 方法。
- 自己实现 UI 部分。

第一种方式虽然可行，但是代码冗余高，实现过于复杂。那第二种方式呢，比如 JUCE 和 Qt 就是用代码实现的。不过在 Qt 的方案中，Android 5 版本或更高版本环境下，很多效果都没法实现，比如按钮没有涟漪效果。根本原因在于它是通过 Qt Quick Controls 的自定义样式来模拟的，而不是使用系统 UI 组件，因此它享受不到系统升级自动带来的界面优化。

当然我们可以使用 OpenGL 来绘制界面，因为 EGL+OpenGL 本身就是跨平台的。并且目前大多数跨平台游戏底层都是这么做的。

既然可以基于 OpenGL 来开发跨平台游戏，那么，是否能用它来进行界面开发呢？当然是可行的，而且 Android 4 的界面就是基于 OpenGL 的，不过它并不是只用 OpenGL 的 API，那样是不现实的，因为 OpenGL API 最初设计并不是为了实现 2D 界面，所以连画个圆形都没有直接的方法，因此 Android 4 中是通过 Skia 将路径转换为位置数组或纹理，然后再交给 OpenGL 从而完成界面渲染的。

然而直接使用 OpenGL 绘制界面，不仅实现的代价大，而且目前支持的平台少。因此对于大多数应用来说自己实现界面是很不划算的。

Xamarin

Xamarin 是从 Mono 发展而来，它用 C# 来开发 Android 及 iOS 应用，因为相关工具及文档都挺健全，因而发展得还不错。在 UI 界面方面，它可以通过调用系统 API 来使用系统内置的界面组件，或者基于 Xamarin.Forms 开发定制要求不高的跨平台 UI。

从实现的方式来讲，iOS 下是以 AOT 的方式编译为二进制文件的；而在 Android 平台上是通过内嵌的 Mono 虚拟机来实现，所以 Xamarin 是跨平台开发的不错选择。

对于熟悉 C# 的团队来说，Xamarin 是一个很不错的方案，但这种方案最大的问题就是相关资料不足，遇到问题很可能搜不到解决方案，并且当前第三方库太少，加之 Xamarin 本身有些 bug，所以让我们静待 Xamarin 做得更好吧。

Go

Go 做为后端服务开发语言，专门针对多处理器系统应用程序的编程进行了优化，使用 Go 编译的程序可以媲美 C 或 C++ 程序的速度，而且更加安全、支持并行进程。Go 从 1.4 版本开始支持开发 Android 应用（1.5 版本支持 iOS）。虽然能同时支持 Android 和 iOS，但是目前可用的 API 很少，Go 语言仍然专注于后端开发。

目前，Android 的 View 层完全是基于 Java 写的，要想用 Go 来完成界面的开发不可避免要调用 Java 代码，而在这方面 Go 还没有简便的实现方式，目前 Go 调用外部代码只能使用 Cgo，通过 Cgo 再调用 jni，这就不可避免地需要写很多的中间件。而且 Cgo 的实现本身就对性能有损失，除了各种无关函数的调用，它还会锁定一个 Go 的系统线程，会影响

其他 goroutine 的运行，如果同时运行太多外部调用，甚至会导致所有 go 线程处于等待状态。

所以，目前使用 Go 开发跨平台移动端应用并不靠谱。

1.2.4 虚拟机流

编译流是将代码编译为不同平台下的二进制文件，而另一种更彻底的做法是：通过虚拟机来支持跨平台运行，比如 JavaScript 和 Lua 都是天生的内嵌语言。不过使用虚拟机进行跨平台开发最普遍的两个问题是：性能损耗；虚拟机本身也会占据不小的空间。

Java虚拟机

说到虚拟机，大家肯定首先想到的是 Java，因为 Java 一开始就是为跨平台而设计的，Sun 的 J2ME 早在 1998 年就有了，在 iPhone 手机出来之前，很多小游戏都是基于 J2ME 开发的。前几年，微软为了支持移动端项目的发展，提供了一套将 Android 和 iOS 代码快速转移到 Windows Phone 的工具，不过后来不了了之。

前面提到 C# 和 Java 在 iOS 端的方案都是通过 AOT 的方式实现的，目前还没见到有 Java 虚拟机相应的方案，主要原因是 iOS 方面的限制。

Titanium/Hyperloop

Titanium 和 PhoneGap 几乎是同时期的著名跨平台方案，和 PhoneGap 最大的区别是：它的界面没有使用 HTML/CSS，而是自己设计了一套基于 XML 的界面框架 Alloy。Titanium 的代码风格如下：

```
APP/styles/index.tss
".container": {
  backgroundColor:"white"
},
// This is APPLIED to all Labels in the view
"label": {
  width: Ti.UI.SIZE,
  height: Ti.UI.SIZE,
  color: "#000", // black
  transform: Alloy.Globals.rotateLeft // value is defined in the alloy.JS file
},
// This is only APPLIED to an element with the id attribute assigned to "label"
"#label": {
  color: "#999" /* gray */
}
```

虽然学习成本低，但 Titanium 同样面临着其他跨平台框架都存在的挑战：缺乏第三方库支持、对外的 API 较少。Titanium 也意识到了这个问题，所以目前在开发下一代的解决方案 Hyperloop，它可以将 JavaScript 编译为原生代码，这样开发者可以方便地调用原生 API。比如调用 iOS 的写法如下：

```
@import("UIKit");
@import("CoreGraphics");
var view = new UIView();
view.frame = CGRectMake(0, 0, 100, 100);
```

这个方案和之前讲到的 Xamarin 如出一辙，也是将 JavaScript 翻译为 Objective-C，然后交由官方系统运行。不过这个项目已经开发了快三年了，但至今仍然是试验阶段，笔者不建议尝试。

React Native

React Native 是由 Facebook 开源的基于 JavaScript 和 React 搭建的一套跨平台开发框架。在设计之初，React Native 采用的方案就是不同平台下使用平台自带的 UI 组件来完成界面的绘制，再加上它采用 JavaScript 和 React 等前端语言来开发，所以获得了不少前端程序员的青睐。

有人说，React Native 采用 JS 等前端技术来开发移动 APP 是回归 H5，但其实 React Native 和 Web 扯不上太多关系，React Native 虽然借鉴 CSS 中的 Flexbox、navigator、XMLHttpRequest 等 API 的写法，但是大部分还是通过原生的组件或者自己封装的组件来开发的。就像 Facebook 的内部软件 Facebook Groups，iOS 版本很大一部分基于 React Native 开发，其中用到了不少内部通用组件。

React Native 相比传统原生开发，学习成本还是比较低的，熟悉 JavaScript 的开发可以迅速实现界面，而使用标签加 CSS 样式表方式绘制的界面，远比原生使用代码绘制的界面更加易读，并且一套界面同时满足 Android 和 iOS 平台，这对于讨厌绘制界面的开发者来说是多么的诱惑。再加上 React Native 师出名门，截至目前，React Native 已更新到 0.4.4 版本，并且趋于稳定。由于其更加接近原生的体验，国内一些大厂纷纷加入，诸如阿里、腾讯、美团等纷纷开始使用 React Native 改造一些应用型 APP。

所以，不管是对于个人还是团队，现在跨平台开发做得最好的就是 React Native，并且随着开源力量的加入，React Native 会发展得越来越好。

1.3 小结

使用 React Native 开发跨平台移动 APP 是一个令人振奋的事情，相比其他跨平台方案，React Native 在不牺牲用户体验和应用质量的前提下，提高了开发效率，使用一套代码即可实现在 Android、iOS 和 Web 平台上运行，节约了成本得到了广大移动开发者的追捧。

本章主要从 React Native 的发展历程和工作原理等方面对 React Native 做了一个简单的介绍，并横向比较了当前主流的跨平台方案。纵观目前的跨平台方案，你会发现 React Native 是目前最好的跨平台技术，如果你的团队正在进行跨平台开发或者考虑跨平台开发，不妨试试 React Native。

React Native 环境搭建与调试

2.1 React Native环境搭建

工欲善其事，必先利其器，在进行正式开发之前，首先需要搭建好相关的开发环境。搭建 React Native 开发环境，需要如下软件。

- Node.js 4.0 及以上版本。
- Mac 系统下 Xcode 6.4 以上，Java 环境及 Android 环境。
- Mac 系统下建议安装 Homebrew，以及 Watchman 和 Flow 等工具。

2.1.1 Mac 环境下搭建 React Native

安装 Node.js

从官网下载 Node.js（官网地址：<https://nodejs.org/en/>），如图 2-1 所示。建议设置 npm 镜像以加速后面的过程。node 安装命令如下：

```
npm config set registry https://registry.npm.taobao.org --global  
npm config set disturl https://npm.taobao.org/dist --global
```



图2-1 Node.js下载与安装

安装完成后，我们输入 `node -v`，如果输出版本号，说明安装成功，如图 2-2 所示。

```
Last login: Fri Mar 31 09:07:26 on console
xiangzhihongdeMacBook-Pro:~ xiangzhihong$ node -v
v7.3.0
xiangzhihongdeMacBook-Pro:~ xiangzhihong$
```

图2-2 查看Node版本

安装npm

npm 是一个包管理工具，用来管理 React 中依赖的包，可以理解为一个项目结构的管理工具，作用类似于 maven、gradle 等。安装命令如下：

```
npm install -g react-Native-cli
```

如果已经安装过，可以使用如下命令进行升级：

```
npm install npm@latest -g
```

安装完 Node 后建议设置下 npm 镜像（淘宝镜像）以加速项目的构建，镜像命令如下：

```
npm config set registry https://registry.npm.taobao.org --global
npm config set disturl https://npm.taobao.org/dist --global
```

安装Yarn

Yarn 是 Facebook 提供的替代 npm 的包管理工具，可以加速 node 模块的下载和构建。相比 npm，Yarn 具有速度快，离线模式，版本控制等优势。安装 Yarn 的命令如下：

```
npm install -g yarn
```

使用 Yarn 下载 React Native 的命令如下，具体如图 2-3 所示。

```
npm install -g yarn react-Native-cli
```

上面的两种方式，不管哪种都可以下载 React Native 环境依赖包，然后静静地等待下线即可。

```
> spawn-sync@1.0.15 postinstall /usr/local/lib/node_modules/yarn/node_modules/spawn-sync
> node postinstall

/usr/local/lib
├── react-native-cli@1.2.0
├── chalk@1.1.3
├── ansi-styles@2.2.1
├── escape-string-regexp@1.0.5
├── has-ansi@2.0.0
├── ansi-regex@2.0.0
├── strip-ansi@3.0.1
├── supports-color@2.0.0
├── minimist@1.2.0
├── prompt@0.2.14
├── pkginfo@0.4.0
├── read@1.0.7
├── mute-stream@0.0.6
├── revalidator@0.1.6
└── util@0.2.1
```

图2-3 使用Yarn下载React Native

安装Homebrew

Homebrew 用于在 Mac 上安装一些 OS X 没有的 UNIX 工具（比如著名的 wget）。React

Native 包管理器同时使用了 node 和 watchman 环境，所以上面的软件还可以使用 Homebrew 来安装，命令如下：

```
brew install node
brew install flow
...
```

安装 Watchman

Watchman 是由 Facebook 提供的监视文件系统变更的工具，此工具可以快速捕捉文件的变化从而实现实时刷新，以便提高开发的性能。Watchman 的安装命令如下：

```
brew install watchman
```

至此，React Native 的基础环境就搭建好了，为了将应用部署到移动设备上，还需要配置 Android、iOS 相关的运行环境。

React Native 版本升级

React Native 作为开源的移动跨平台框架，在很多细节上还不是很完善，所以版本迭代更新的速度相对较快。为了更好地开发出高质量的移动应用，需要不断地对 React Native 版本进行升级。

首先，使用命令查看本地的版本，命令如下：

```
react-native --version
```

然后，使用 npm 包管理命令查看 React Native 版本信息，命令如下，具体如图 2-4 所示。

```
npm info react-native
```



图2-4 查看版本信息

使用升级命令直接升级到指定版本，如下：

```
npm install --save react-native@0.45
```

升级完成之后，最常见的错误就是依赖的模块版本不统一，为此，根据提示升级 React 等相关模块即可。然后运行如下命令：

```
react-native upgrad
```

对于一些老项目，版本跨度比较大，建议对项目进行备份，将老项目的相关逻辑代码拷贝到新建的项目中，拷贝的内容大体如下：

iOS 文件目录

- ProjectName.xcodeproj
- AppDelegate.h
- AppDelegate.m
- Info.plist
- Images.xcassets
 - Image.imageset
 - AppIcon.appiconset
 - Contents.json

Android 文件目录

- android/settings.gradle
- android/app/build.gradle
- android/app/proguard-rules.pro
- android/app/src

至此，React Native 的升级操作就完成了。假如升级到最新版本后不是特别稳定，想要回到之前的版本，有两种方式：第一种是修改 package.json 文件；第二种是执行命令 `npm install --save react-native@版本号`。经过上述操作之后，不要忘了更新相关的依赖文件，如果降级之后报错，建议删除 node-modules 文件夹，然后在使用 `npm install` 命令重新更新依赖文件。

2.1.2 React Native开发IDE

安装Atom

Atom 是专门为软件开发工作者推出的一款跨平台文本编辑器，如图 2-5 所示。它具有简洁和直观的图形用户界面，并有很多有趣的特点：支持 CSS、HTML 和 JavaScript 等网页编程语言；支持宏，自动完成分屏功能，集成了文件管理器等功能和优点。

打开 Atom 的官网 (<https://atom.io/>)，当然你也可以到国内的镜像地址下载，比如淘宝国内镜像地址：<https://npm.taobao.org/mirrors/atom/1.7.2/>

React Native 官方推荐 Atom+Nuclide 来开发。Atom 集成 Nuclide 的步骤如下：

点击菜单栏【Atom】→【Preferences】，或者使用快捷键【command+,】打开设置面板，然后在 Install Packages 的输入框中，输入 Nuclide，点击【install】，如图 2-6 所示。安装完成之后就可以在 Atom 的工具栏看到 Nuclide 插件，如图 2-7 所示。

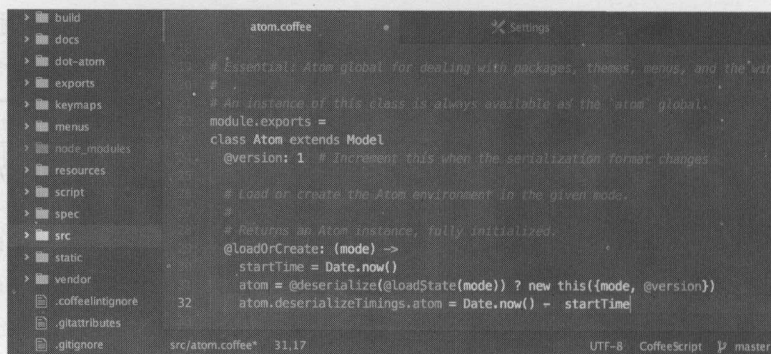


图2-5 Atom编辑器

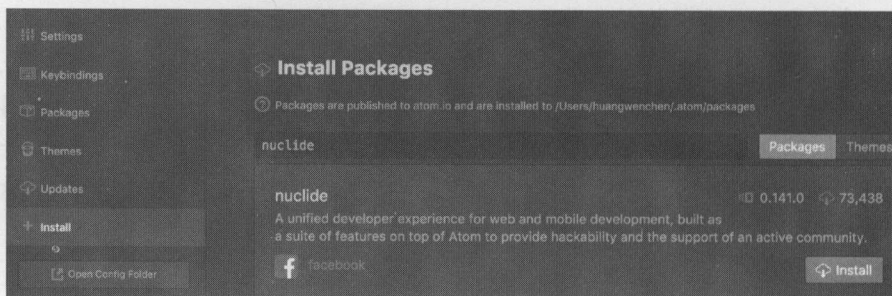


图2-6 Atom安装Nuclide插件

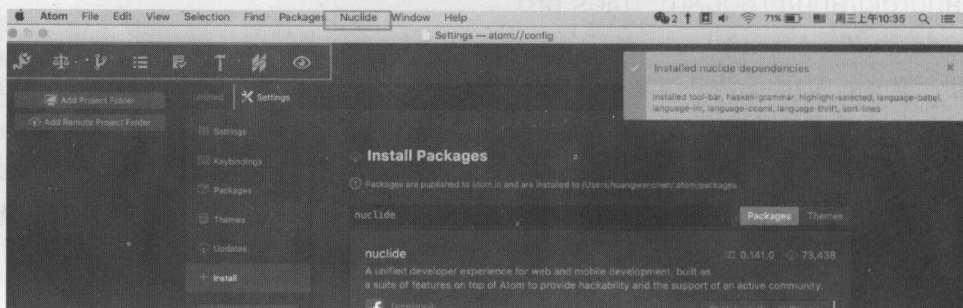


图2-7 Nuclide安装结果

到此，React Native开发所需的IDE环境就安装完成了，接下来就可以使用IDE编写代码了。

2.1.3 创建React Native项目

在创建项目方面可以使用IDE辅助创建React Native项目，也可以使用命令方式创建。使用命令方式创建项目，首先创建项目工作空间，然后打开Mac终端，输入如下命令创建项目，如图2-8所示：

```
react-native init Demo (项目名)
```

然后等待 npm 构建项目即可。

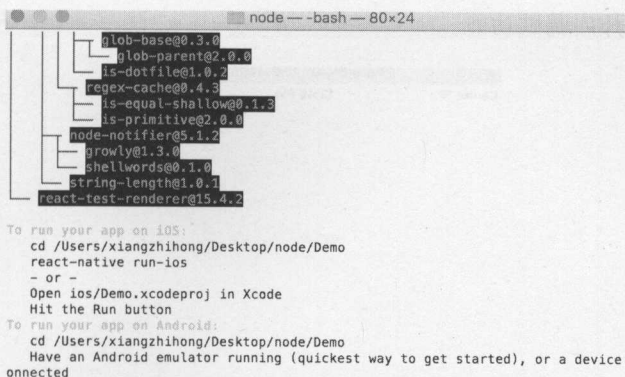


图2-8 使用命令创建项目

当项目构建完成之后，使用 Atom 导入新建的项目就可以开始编写代码了。如果使用 WebStorm 作为 React Native 开发 IDE，则比较简单。

2.1.4 运行React Native项目

使用快捷键【command+shift+p】打开面板，运行命令“React Native Packager”，或者使用图形化界面，依次点击【Nuclide】→【React Native】→【Start Packager】启动打包操作，如图 2-9 所示。

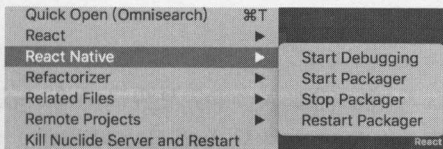


图2-9 启动React Native

打开终端运行项目，先定位到项目所在的位置，然后使用如下命令运行项目：

```
// 运行 iOS
react-native run-ios
// 运行 Android
react-native run-android
```

如果要在真机上运行 iOS 应用，则需要准备一台装有 Mac 系统的电脑，同时还需要一个 Apple ID。如果需要把应用发布到 App Store，那么还得去苹果开发者网站购买一个开发者账户（在自己手机上测试则不用）。要想在真机上运行 iOS 应用，你可以将 USB 连接至电脑，或者在 Xcode 的设备管理中添加你的设备。然后你就可以在你的真机上使用 iOS 应用或者调试应用了，如图 2-10 所示。

Android 设备只需要使用 USB 安装即可。如果需要使用真机调试，摇晃设备就可以打开开发者菜单，将其中的【localhost】改为电脑的 IP 地址，启用开发者菜单中的【Debug JS

Remotely】选项即可开启调试功能。



图2-10 运行iOS项目

2.1.5 iOS环境

目前 React Native 在 iOS 环境下需要 Xcode7 及更高版本, 使用 Xcode 打开 React Native 项目的 iOS 文件夹中, 选中 .xcodproj 文件导入项目文件即可, 如图 2-11 所示。导入后的项目结构如图 2-12 所示。

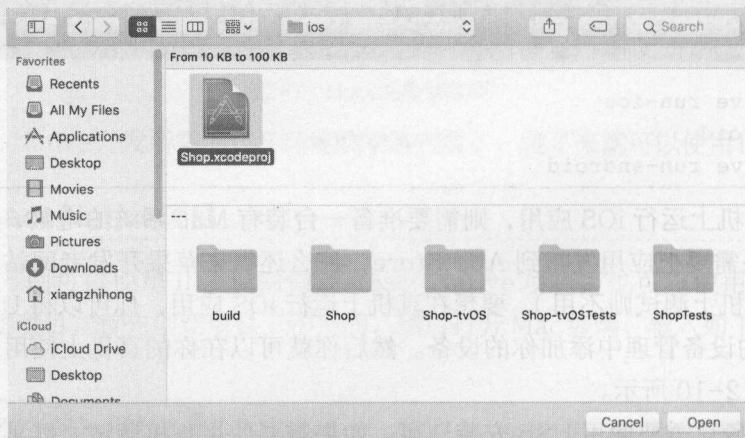


图2-11 导入iOS项目

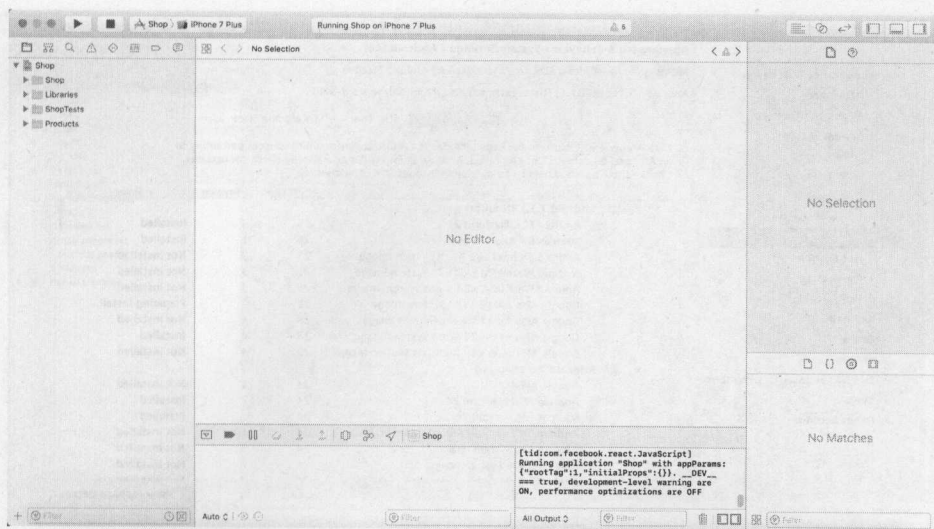


图2-12 iOS项目结构

使用快捷键【command+R】或者直接点击运行按钮运行项目可能会报错，这时候你需要使用命令先启动 React Native 相关依赖环境。启动命令如下：

```
react-native start
```

2.1.6 Android环境

相比 iOS 的环境依赖，Android 的环境依赖就复杂得多，对应的官方配置文档地址为：<http://facebook.github.io/react-native/docs/android-setup.html>。配置 Android 依赖环境，总体来说分为三步：安装 Java 环境，安装 Android SDK 环境，配置其他 Android 环境。

首先，需要安装配置 Java 和 Android 环境。

❶ 下载并安装最新版本的 JDK，请注意选择的是 x86 还是 x64 版本，JDK 的 bin 目录加入系统 PATH 环境变量中。JDK 下载地址为：<http://www.oracle.com/technetwork/java/javase/downloads/index-JSp-138363.html>。

❷ 安装 Android SDK 环境，可以单独安装 Android SDK，也可以通过 Android Studio 一并安装。这里直接使用 Android SDK 一并安装。Android Studio 的下载地址为：<http://www.android-studio.org/>。然后配置 Android 相关环境变量。

❸ 打开 Android SDK 管理器，如图 2-13 所示。确保选中以下选项。

- Android SDK Build-tools version 23.0.1
- Android 6.0 (API 23)
- Android Support Repository

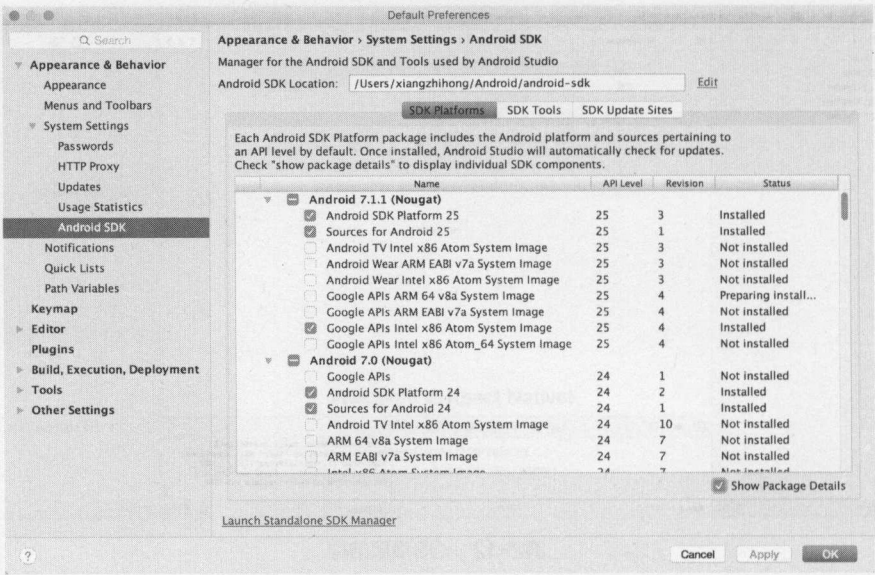


图2-13 Android SDK管理器

接下来安装模拟器，推荐使用 Genymotion，这是一款强大的 Android 模拟器，下载地址：<http://www.genymotion.net/>。在 Android Studio 配置 Genymotion 插件方面，使用快捷键【command+,】，或者依次点击【Android Studio】→【Preferences】→【Plugins】，搜索 Genymotion 插件并安装，如图 2-14 所示。安装完成之后，打开 Android studio，点击 Genymotion 插件图标配置相关信息即可。

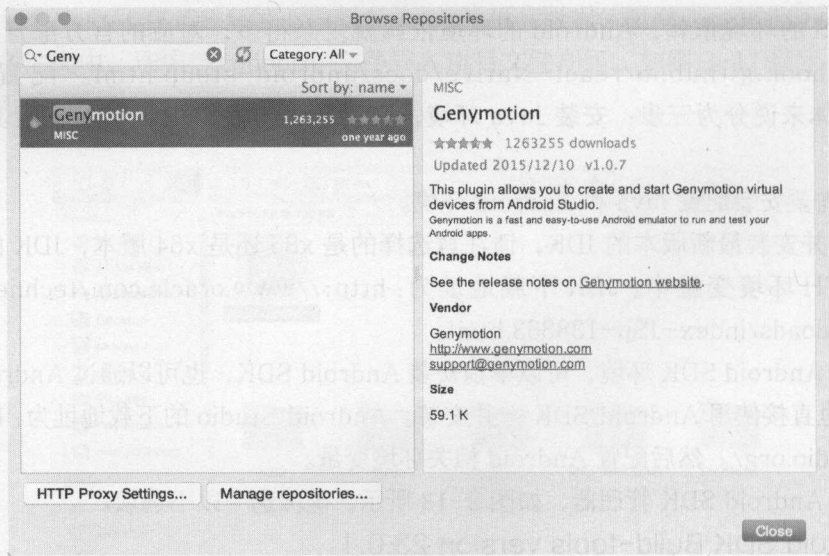


图2-14 Android Studio安装Genymotion插件

打开 Android Studio 导入 React Native 项目下的 Android 文件，如图 2-15 所示。

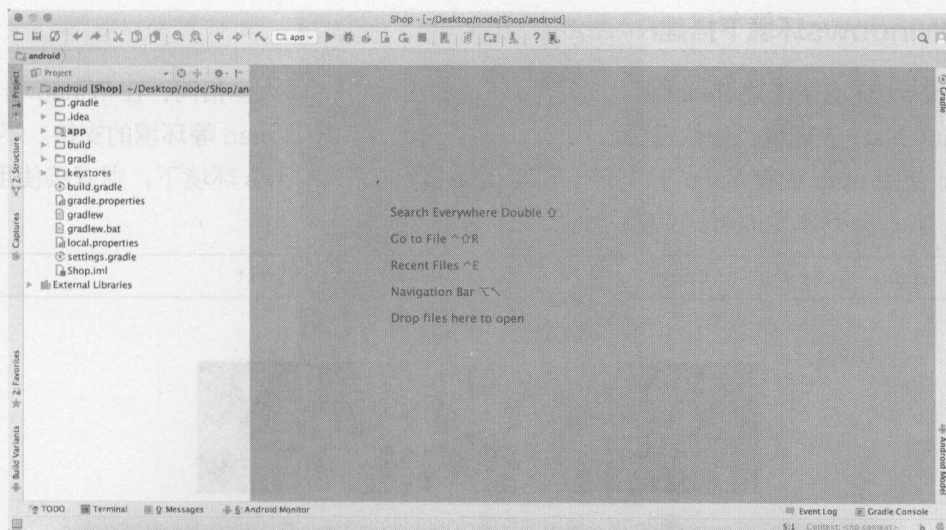


图2-15 导入Android项目

定位到项目目录，使用命令 `npm start` 启动 React Native 相关依赖环境，然后使用命令 `react-native run-android` 启动 Android 项目即可，如图 2-16 所示（注意，运行命令前先启动 Android 模拟器）。

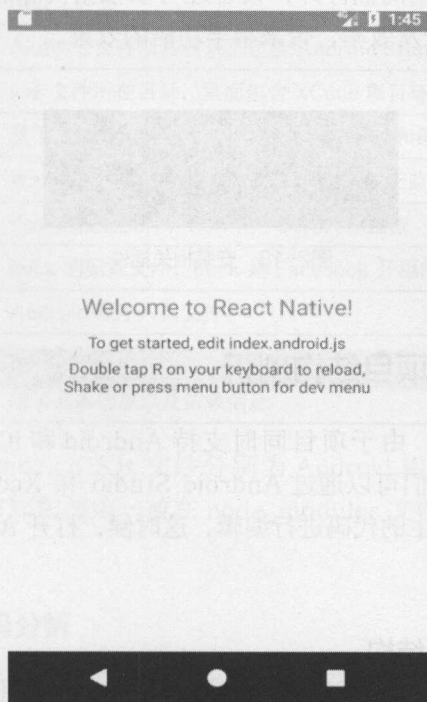
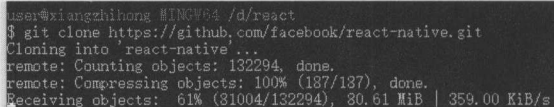


图2-16 Android运行React Native项目

2.1.7 Windows环境下搭建React Native

Windows 环境下搭建 React Native 开发环境的步骤和 Mac 大体相同，在 Windows 环境下搭建 React Native 环境，依然需要 Node、npm、Yarn、Watchman 等环境的支持。不同之处在于 Mac 使用 npm 或者 Yarn 下载 React Native，而在 Windows 环境下，则直接使用 git 克隆 React Native 到本地即可。克隆命令如下，具体见图 2-17。

```
git clone https://github.com/facebook/react-native.git
```



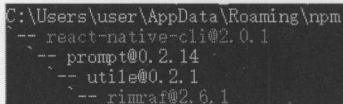
```
user@xiangzhihong MINGW64 /d/react
$ git clone https://github.com/facebook/react-native.git
Cloning into 'react-native'...
remote: Counting objects: 132294, done.
remote: Compressing objects: 100% (137/137), done.
Receiving objects: 61% (31004/132294), 30.61 MiB | 359.00 KiB/s
```

图2-17 使用git克隆React Native到本地

进入 React Native 目录下的 react-native-cli 目录，输入如下命令即可安装相关依赖包到全局环境中。

```
npm install <Name>-g
```

当然，除了如图 2-18 介绍的插件外，读者还可以根据实际需要安装一些其他插件，这些插件能提升 React Native 的开发效率，带来事半功倍的效果。



```
C:\Users\user\AppData\Roaming\npm
-- react-native-cli@2.0.1
-- prompt@0.2.14
-- util@0.2.1
-- rimraf@2.6.1
```

图2-18 安装相关插件

2.2 React Native 项目结构剖析

在 React Native 项目中，由于项目同时支持 Android 和 iOS，所以我们会看到完整的 Android 和 iOS 项目结构。我们可以通过 Android Studio 和 Xcode 来打开相应的项目，在混合开发的时候，往往需要对原生的代码进行编辑，这时候，打开 Android Studio 和 Xcode 编写差异化代码即可。

2.2.1 React Native文件结构

当新建一个 React Native 项目后，使用 IDE 打开后看到的项目目录结构如图 2-19 所示。

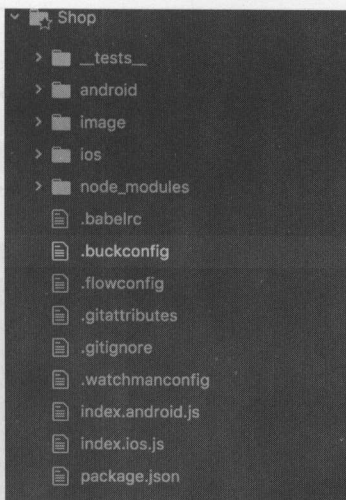


图2-19 React Native项目结构

React Native 项目文件组成及相关说明见表 2-1。

表 2-1 React Native 文件表

名称	功能描述
__tests__	测试文件夹
android	Android 文件所在目录，包含 Android Studio 项目环境文件
iOS	iOS 文件所在目录，里面包含 XCode 项目环境文件
node_modules	基于 node 文件依赖系统产生的相关依赖和第三方 lib
watchmanconfig	Watchman 的配置文件，Watchman 用于监控文件变化
flowconfig	flow 的配置文件，flow 用于代码静态检查
buckconfig	buck 的配置文件，buck 是 Facebook 开源的高效编译系统
index.android.js	Android 程序入口文件
index.ios.js	iOS 应用入口文件
package.json	项目基本信息以及依赖信息

其中 index.android.js 和 index.iOS.js 文件分别为 Android 和 iOS 的启动入口文件。React Native 项目中所依赖的第三方库则被统一放在 node_modules 文件夹下，由 package.json 进行统一管理。

2.2.2 iOS文件结构及代码分析

React Native 项目中，同时包含 Android 和 iOS，iOS 的项目结构和原生的项目结构一致，其项目结构如图 2-20 所示。

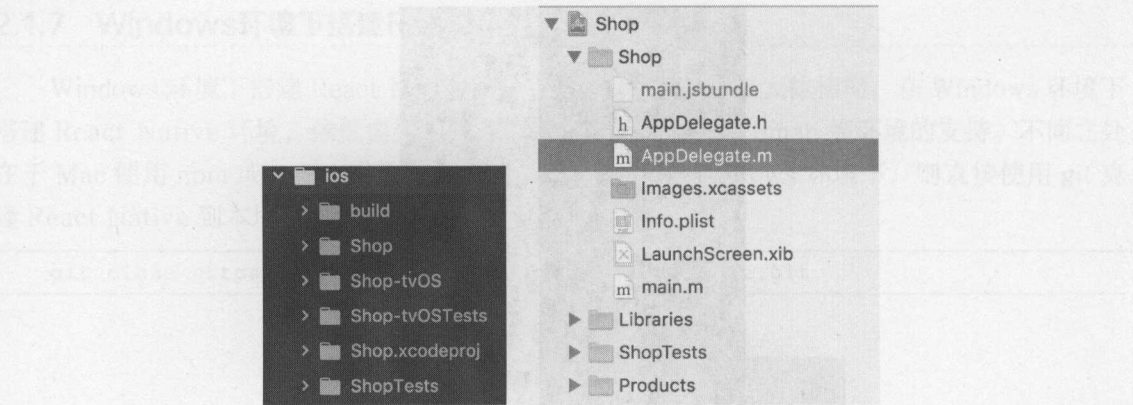


图2-20 iOS项目结构

React Native 文件 iOS 目录如表 2-2 所示。

表 2-2 iOS 文件表

名称	功能描述
build	项目的编译文件
shop	iOS 对应项目所在的位置
shop-tvOS	tvOS 对应的项目文件
xcodeproj	iOS 项目工程启动文件

或许，读者会有疑问，在 index.ios.js 注册启动文件后，iOS 是如何启动原生视图的呢？答案在 AppDelegate.m 文件中。打开 AppDelegate.m，在 AppDelegate.m 中声明的根视图中有如下代码：

```
RCTRootView *rootView = [[RCTRootView alloc]
initWithBundleURL:JSCodeLocation
moduleName:@"Shop"
initialProperties:nil
launchOptions:launchOptions];
```

React Native 库将其所有的类名使用 RCT 作为前缀，也就是说 RCTRootView 其实是 React Native 的类。而在 iOS 中，RCTRootView 表示 React Native 的根目录。AppDelegate.m 通过将视图添加到 UIViewController 中并渲染到屏幕上。例如，在本示例项目中，打开 index.ios.js，在最后一行看到代码中暴露 Shop 组件，从而完成渲染工作。相关代码如下：

```
APPRegistry.registerComponent('Shop', () => Shop);
```

2.2.3 Android文件结构及代码分析

在 React Native 中，Android 的项目结构和原生应用项目结构是一致的，其项目目录结

构如图 2-21 所示。

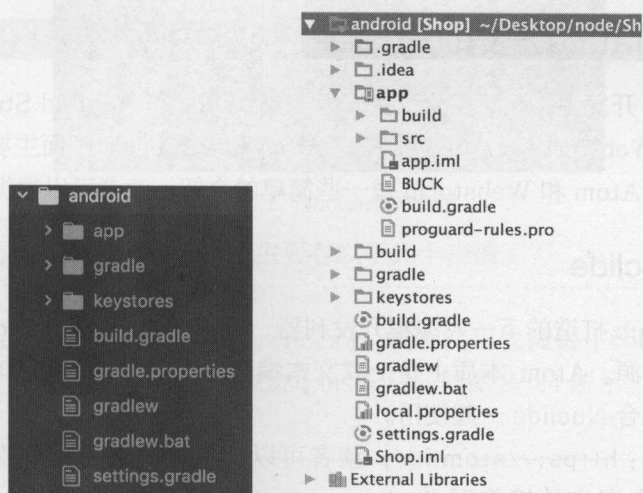


图2-21 Android项目结构

Android 项目文件及说明如表 2-3 所示。

表 2-3 Android 文件表

名称	功能描述
app/build	app 模块 build 编译输出的目录
app/build.gradle	app 模块的 gradle 编译文件
app/app.iml	app 模块的配置文件
app/proguard-rules.pro	app 模块的 proguard 文件
build.gradle	项目的 gradle 编译文件
settings.gradle	项目中的配置文件
gradlew	脚本文件，可以在命令行执行打包
local.properties	本地配置信息，如 sdk、ndk 配置
External Libraries	项目依赖的 Lib，编译时自动下载

index.android.js 作为 Recott Ncotive 项目 Andrad 端的启动入口，是如何启动原生视图的呢？正如 AppDelegate.m 之于 iOS，Android 的入口在 MainActivity 文件中，其核心的方法在 getMainComponentName() 中：

```
protected String getMainComponentName() {
    return "Shop";
}
```

除了上面介绍的文件外，React Native 项目中所依赖的包都会统一放在 node_modules 文

件夹下。关于使用 React Native 进行混合开发过程中的一些细节，后面章节将慢慢介绍。

2.3 React Native开发IDE介绍

在 React Native 开发中，除了需要原生开发工具 Xcode 和 Android Studio 之外，推荐的开发工具有 Sublime、Webstorm 以及官网推荐的 Atom 和 VSCode。下面主要对 React Native 开发中用到的开发工具 Atom 和 Webstorm 做一些简单的介绍，读者可以根据喜好自行选择。

2.3.1 Atom+Nuclide

Atom 是由 Github 打造的下一代编程开发利器，支持 Windows、Mac OS X、Linux 三大桌面平台，免费且开源。Atom 本质上是一款文本编辑器，而不是一款 IDE，所以使用它开发 React Native 需要配合 Nuclide 一起使用。

Atom 的官网为：<https://atom.io/>，读者可以到官网下载 Atom 安装包，如果网速较慢，还可以访问国内淘宝的镜像网站：<https://npm.taobao.org/mirrors/atom/1.7.2/>，点击相应的版本下载并安装，如图 2-22 所示。

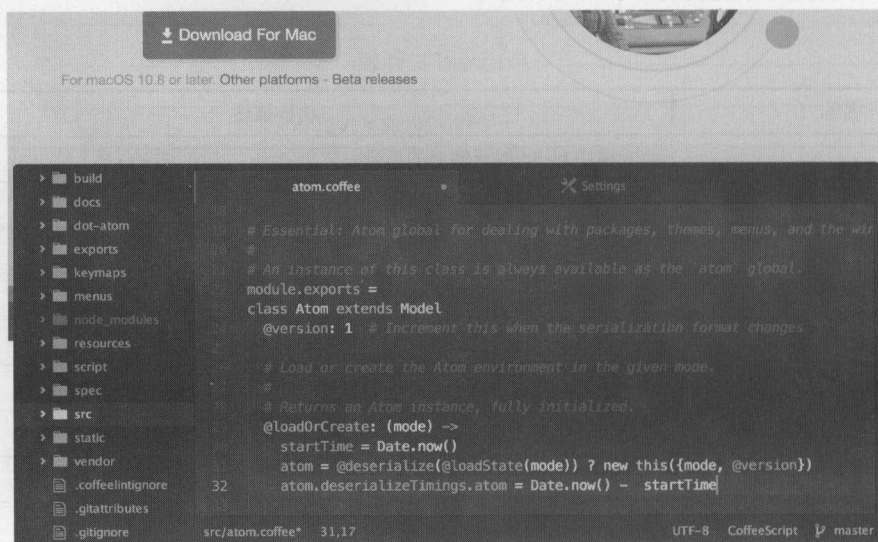


图2-22 Atom官网

Nuclide 是 Facebook 在 Atom 的基础上开发的一款插件 IDE，可以用来开发 React Native、iOS 和 Web 应用，目前暂不支持 Windows，只支持 Mac OS X 和 Linux 系统环境。Nuclide 内置了对 React Native 的支持，包括代码自动补全、代码诊断等功能。而官方也推荐使用 Atom+Nuclide 来开发 React Native 应用。

Nuclide 项目官方地址为：<https://github.com/facebook/nuclide>，打开后便可看到相关介绍，如图 2-23 所示。

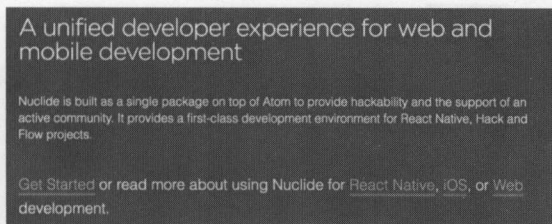


图2-23 Nuclide官方介绍

在 Atom 中安装和使用 Nuclide 插件主要有以下几个步骤。

安装Nuclide

❶ 点击菜单栏:【Atom】→【Preferences】, 或者使用快捷键【command+,】在 Install Packages 的输入框中输入 nuclide, 然后点击【install】, 如图 2-24 所示。当然也可以使用命令安装, 命令如下:

```
apm install nuclide
```

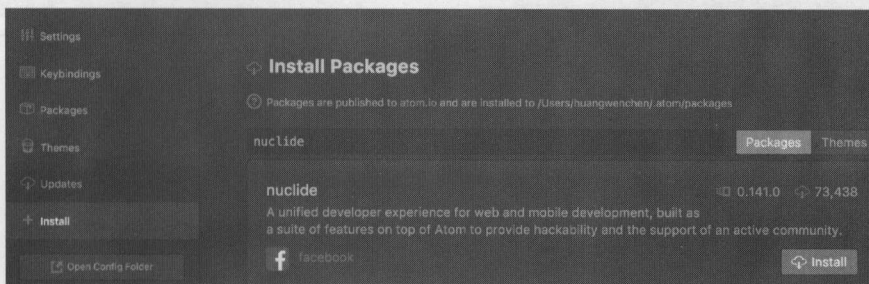


图2-24 安装Nuclide插件

❷ 安装完成之后, 将在 Atom 的工具栏中看到 Nuclide 标签, 如图 2-25 所示。

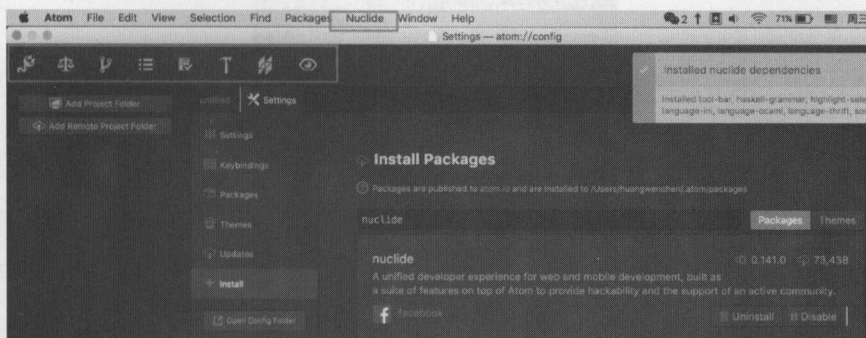


图2-25 Nuclide插件

❸ 安装 Nuclide 之后, 会安装一大堆依赖包, 如果默认没有安装, 可以手动安装。依次选择【Packages】→【Settings View】→【Manage Packages】, 如图 2-26 所示。搜索 nuclide, 进入设置, 勾选【Install recommended packets on startup】即可。

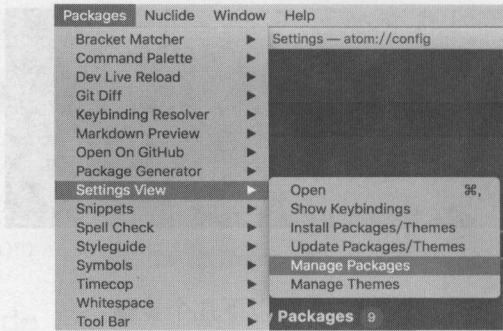


图2-26 Nuclide依赖包管理

使用Nuclide运行项目

项目开发完成后，启动 React Native 项目主要有两种方式：一种方式是使用命令方式启动，而另一种方式就是直接利用 IDE 的相关图形化界面启动。使用命令方式启动的步骤如下，使用快捷键【command + shift + p】打开终端面板【command palette】，输入如下打包命令，具体如图 2-27 所示。

```
react native start packer
```

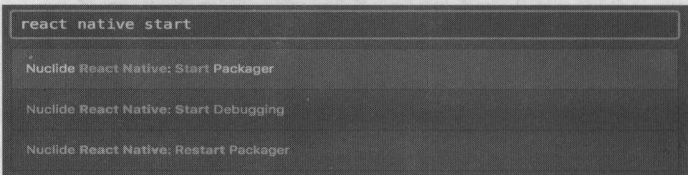


图2-27 命令方式启动项目

如果使用 Nuclide 的图形化界面来运行项目，则可以依次选择【Nuclide】→【React Native】→【Start Packger】选项即可，如图 2-28 所示。

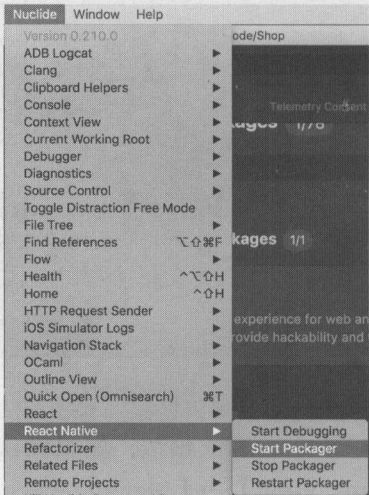


图2-28 图形界面方式启动项目

不管使用哪种方式，都可以启动 React Native，上述方式无需定位到项目目录，也不涉及任何命令，相比命令方式，使用 Nuclide 启动 React Native 更加方便。

2.3.2 WebStorm

WebStorm 是 JetBrains 公司开发的 Javascript 集成开发环境，可以用于客户端应用开发以及 Web 开发，目前，WebStorm 支持 Windows、Mac OS X、Linux 三大桌面平台。最新版的 WebStorm 还支持界面化创建项目、界面化运行及调试功能，这对于习惯了 Android Studio 和 Xcode 界面化开发的人来说的确是不小的惊喜。

WebStorm 的官网地址为：<https://www.jetbrains.com/Webstorm/>，下载相应的版本安装即可，如图 2-29 所示。需要注意的是，WebStorm 只支持 30 天的试用时间，可以使用“license server”方式激活，激活 server 地址为：<http://idea.imsxm.com/>。



图2-29 WebStorm下载

最新版本的 WebStorm 默认支持 JSX 语法，并且其拥有强大的插件库，如代码提示和自动补全插件（ReactNative-LiveTemplate）、代码格式化插件以及配置 WebStorm 启动应用插件等。对于习惯了 idel 系列工具的开发来说，使用 WebStorm 来开发 React Native 会是一个不错的选择。

除此之外，WebStorm 在最新版本中默认添加了对 React Native 环境的支持，开发者可以直接使用 WebStorm 新建项目（见图 2-30）、运行项目和调试项目（见图 2-31）等。同时，WebStorm 还有强大的日志系统，我们可以使用 WebStorm 自带的控制台查看相关日志，这对于我们理解应用的运作也是非常有好处的。

除了提供强大的插件库和默认对 React Native 的支持之外，WebStorm 也支持使用图形化界面运行、调试应用，其强大的代码提示功能对于习惯了 Xcode 和 Android Studio 的 APP 开发者来说，可谓得心应手。

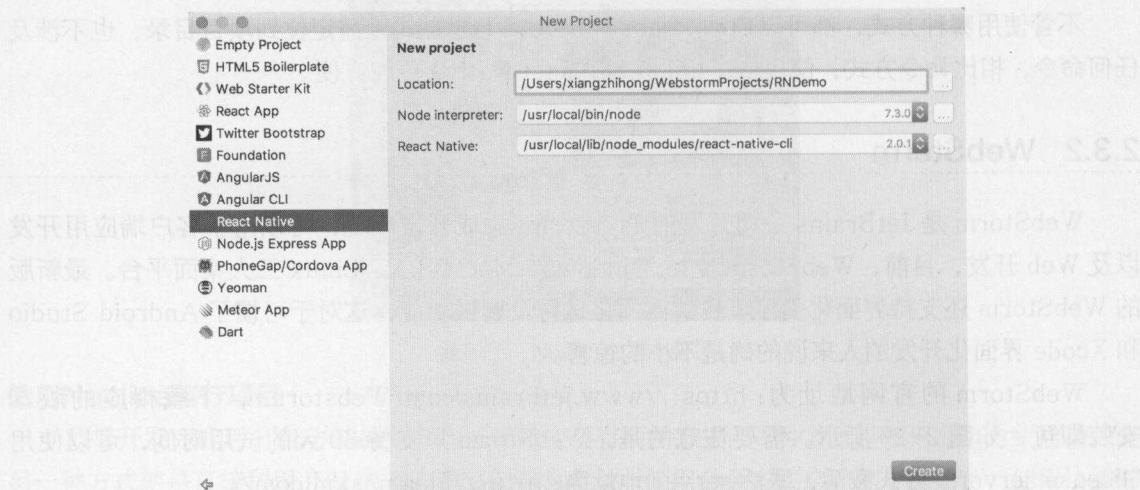


图2-30 使用WebStorm创建项目

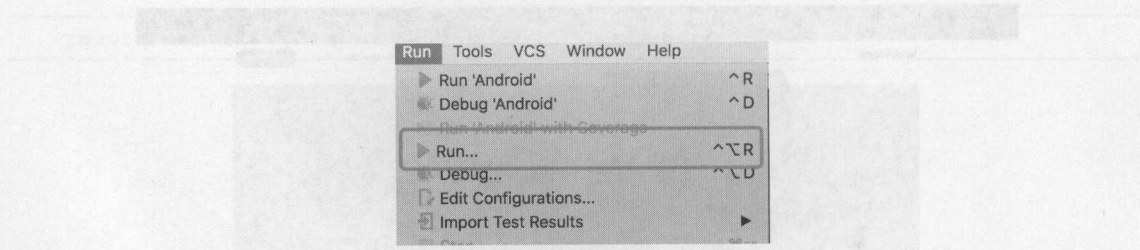


图2-31 使用WebStorm运行、调试项目

2.4 React Native调试技巧

在项目开发过程中，由于各种原因，遇到的问题往往是不可预知的，为了更好地发现程序中的 bug，软件开发人员可以通过对程序设置断点来定位问题，进而修复 bug。

2.4.1 JavaScript调试技巧

当我们使用 JavaScript 开发 Web 产品时，有一些常用的调试 JavaScript 的技巧和工具，它们同时也适用于 React Native 开发中的调试工作，因此了解 JavaScript 调试技巧可以让我们更快速地调试 React Native 应用程序。

console.log日志

对于大多数的软件开发工作而言，日志调试是不可缺少的代码调试方式之一。在 Web 开发中，在代码中添加 console.log 日志输出是软件开发流程中的一个潜在环节，它可以让开发人员更方便地了解代码的执行流程。

不管是 Xcode (iOS IDE) 还是 Android Studio (Android IDE)，它们的控制台都可以打印我们想要的日志信息，前提是你代码中添加了相关的日志。Xcode 控制台日志如图 2-32 所示。

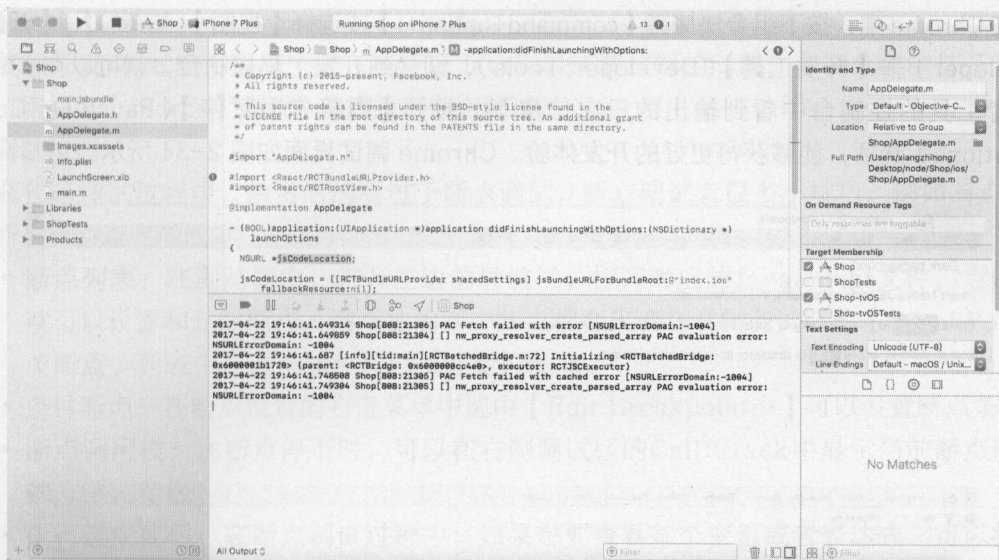


图2-32 Xcode控制台日志

与 Xcode 类似, Android Studio 也提供了类似的日志输出窗口, 并且还可以根据自己的需要进行条件筛选, 如图 2-33 所示。

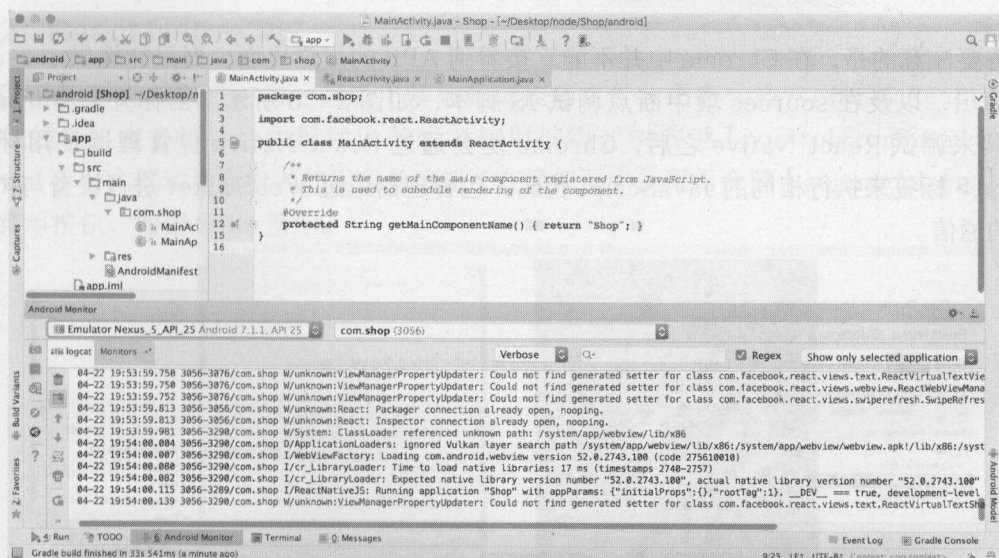


图2-33 Android Studio控制台日志

对于原生开发而言, Xcode 和 Android Studio 是最好的选择, 然而对于 Web 开发而言, 借助浏览器进行代码的调试是一个不可忽略的技巧, 具体使用的时候, 激活开发者菜单, 在开发者菜单选择【Debug in Chrome】, 打开一个新的标签页: <http://localhost:8081/debugger->

UI。在 Chrome 中，按下组合快捷键【command+option+i】或选择【视图】(View) → 【开发者】(Developer) → 【开发工具】(Developer Tools)，切换到开发工具控制台，就可以在 Chrome 开发者工具的控制台中看到输出的日志信息了。打开【有异常时暂停】(Pause On Caught Exceptions) 选项，能够获得更好的开发体验。Chrome 调试界面如图 2-34 所示。

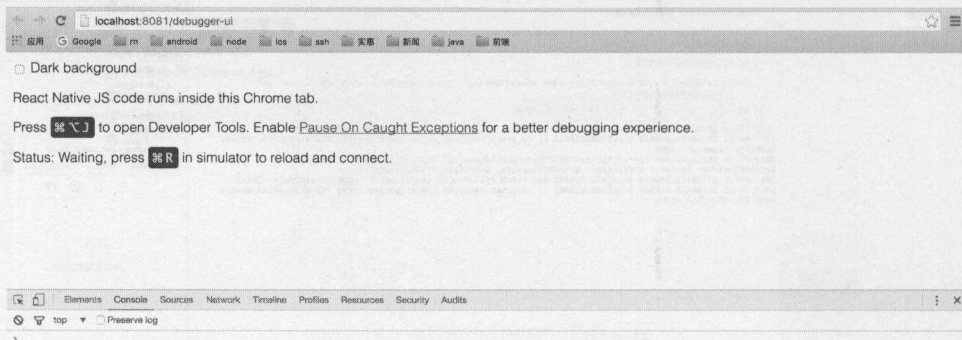


图2-34 Chrome控制台日志

需要注意的是，在 Chrome 中并不能直接看到 APP 的用户界面，而只能提供 console 的日志输出，以及在 sources 项中断点调试 JS 脚本，如图 2-35 所示。当你开启 Chrome 调试工具来调试 React Native 之后，Chrome 便会通过 React Native 包管理器使用标准的 <Script> 标签来执行相同的 JavaScript 代码。包管理器通过 WebSocket 进行设备与浏览器之间的通信。

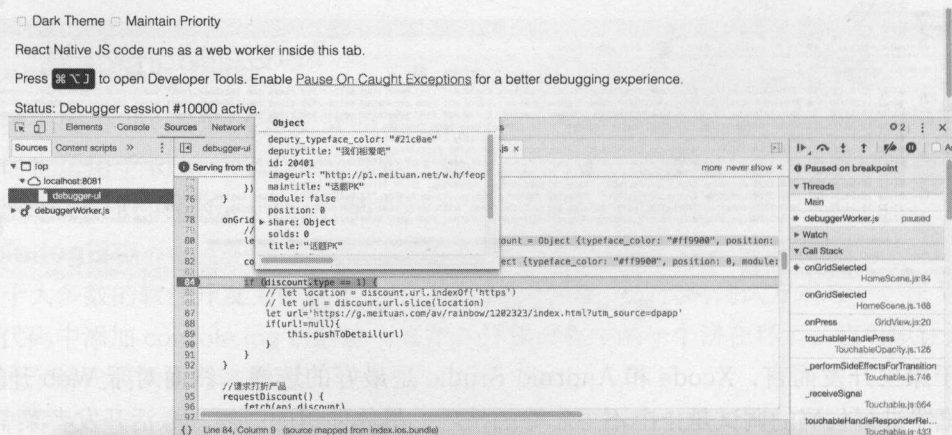


图2-35 Chrome控制台日志

JavaScript调试器

就像开发 Web 应用程序一样,在使用 JavaScript 调试器的时候,在浏览器中打开开发者选项,切换至【source】选项,然后执行断点调试。

断点调试技巧

在代码调试过程中,最常用的莫过于断点调试,断点调试有很多小技巧,例如断点列表、条件断点、断点调用栈等。

- 断点列表:在浏览器中,打开左边的导航栏,打开对应的JavaScript文件,点击行号就可以设置和删除断点,而添加的每个断点都会出现在右侧调试区的【BreakPoints】(断点)列表中,这个功能可以快速定位断点。
- 条件断点:在断点位置的右键菜单中选中【Edit Breakpoint...】可以设置断点条件。
- 断点调用栈:在断点停下时,可以在右侧调试区的Call Stack中显示当前断点所处处的方法调用栈。
- 执行选中代码:在断点调试过程中,如果想要查看某个变量或者表达式,可以将其选中后单击鼠标右键,然后选中【Evaluate in Console】来观看选中值的结果。
- 在DOM元素上设置断点:有时候开发者需要监听某个DOM被修改的情况,例如当数据刷新时DOM元素是否被刷新等,这时候可以在审查的Elements Panel元素的右键菜单里选择断点,当对应的元素发生改变时,就会触发该处断点。

2.4.2 React Native调试

除了基于 JavaScript 的一些通用基础调试技巧之外,还有一些专门针对 React Native 的调试技巧。在 iOS 平台中摇动设备或者使用按组合快捷键【control+command+z】,在 Android 平台中摇动设备或者使用菜单按钮(在模拟器中使用组合键【command +m】)调出硬件菜单按钮,如图 2-36 所示。

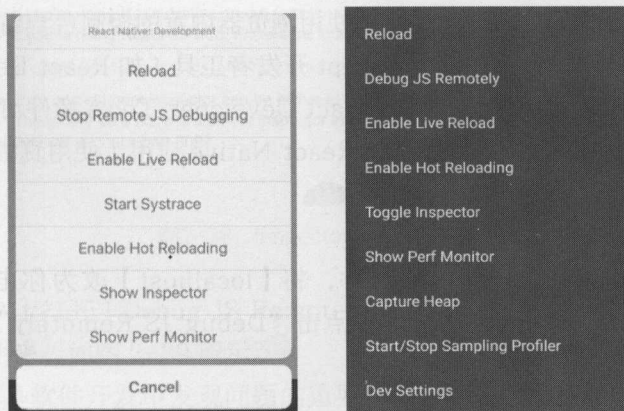


图2-36 Android、iOS调试

需要注意的是,在测试环境中开发者菜单默认是打开的,开发者可以在测试环境进行相关

调试，而在线上环境中开发者菜单会被关闭。

在 iOS 平台上，打开 Xcode 中的项目，选择【Roduct】→【Scheme】→【Edit Scheme】，或使用组合键【command+<】。下一步，在左边的菜单中选择【Run】然后将【Build Configuration】改为【Release】。

在 Android 平台上，默认情况下，由 gradle 建立发布的开发者菜单将被禁用（例如，gradle 的 assembleRelease 任务）。虽然这种行为可以通过传递给 ReactInstanceManager#set UseDeveloperSupport 正确的值来自定义。

启动 React Native 项目，在浏览器中打开一个新的页面：<http://localhost:8081/debugger-ui>，在 Chrome 浏览器中打开开发者工具（其他浏览器类似），或者使用快捷键【command + option + i】切换到控制台页面，如图 2-37 所示。

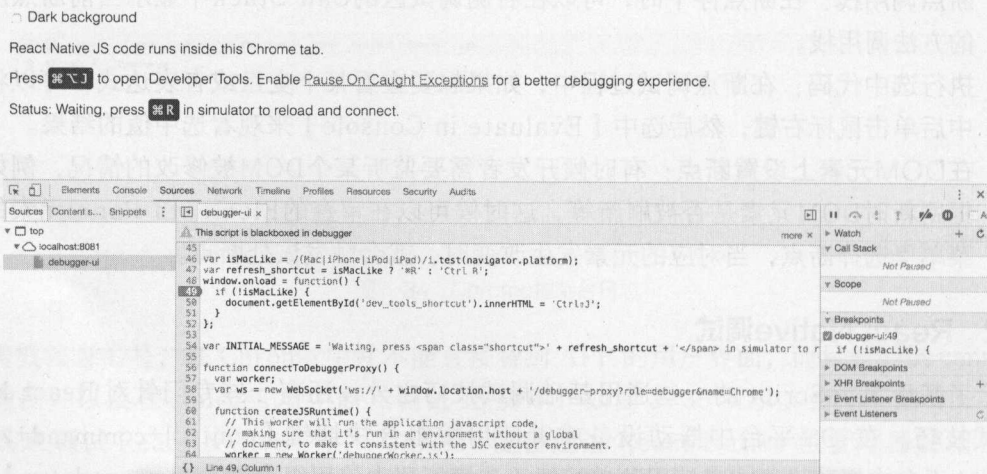


图2-37 Chrome控制台页面

使用 JavaScript 调试器，可以很方便地使用浏览器内置的控制台与当前的 JavaScript 上下文进行交互。在 Web 开发中，使用 JavaScript 开发者工具（如 React Developer Tools 插件）可以查看组件的层次结构、属性和状态，React Developer Tools 插件可以在 Chrome 的扩展程序中安装。当然，在浏览器中也可以调试 React Native 应用（使用真机调试，确保手机和电脑处于同一个局域网），具体步骤如下。

iOS调试

打开 RCTWebSocketExecutor.m 文件，将【localhost】改为你电脑的 IP 地址，然后打开手机或者模拟器在 Developer Menu 下点击“Debug JS Remotely”选项启动 JS 远程调试功能。

Android调试

React Native 调试 Android 程序的方式有两种。第一种，通过在【Developer Menu】下的【Dev Settings】中设置你的电脑 IP 来进行调试；第二种，是命令方式，主要针对

Android5.0 以上设备，将手机通过 USB 连接到你的电脑，然后通过 adb 命令行工具连接手机进行调试，相关命令如下：

```
adb reverse tcp:8081 tcp:8081
```

调试面板介绍

Element 面板在这个面板中，开发者可以通过 Element 面板查看整个页面的 DOM 结构。在调试模式下，还可以查看 CSS 元素、HTML 标签等信息。

- Sources面板：Sources模块包含着各种资源文件，它是按照页面中出现的域来组织的，一些异步加载的JS文件，在加载之后也会出现在这里，当调试时，断点数据和变量也会显示在这里。
- Timeline面板：使用Timeline可以让开发者看到浏览器的渲染过程，进而根据时间轴上的FPS、CPU的占用情况提出相关的优化方案。
- Profiles面板：Profiles主要用来检测CPU的占用程度、堆栈申请的内存情况。Profiles主要用来配合Timeline使用，从而提供更好的优化方案。

Element Inspector命令

利用 Element Inspector 命令软件开发人员可以很方便地查看 React Native 界面的层次结构，使用快捷键【command+shift+p】打开终端，输入 inspector 命令，等待远端连接，如图 2-38 所示。

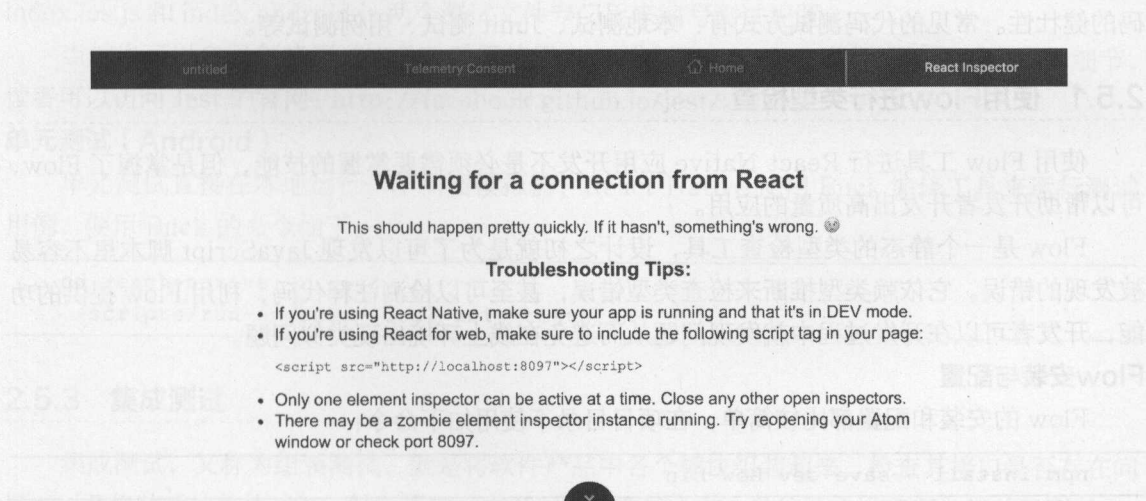


图2-38 Inspector调试等待连接

在客户端的 APP 上打开【Debug JS Remotely】选项重新加载，连接上以后就可以将当前项目的 UI 层次显示出来，如图 2-39 所示。

软件代码调试，是软件开发中发现问题的重要手段，当发生程序逻辑问题时，调试可以帮助开发者快速定位问题并解决问题。而通过使用调试工具，不仅可以快速跟踪程序中出现的问题，还能利用工具提供的性能分析功能来提高程序的性能，从而提高应用的体验。

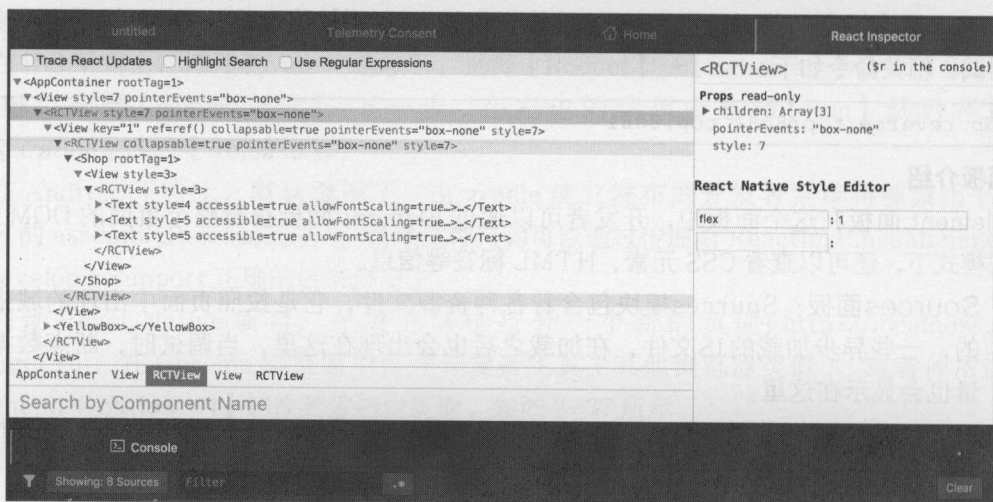


图2-39 Inspector UI层次调试

2.5 React Native代码测试

进行代码调试是提高代码质量的一个重要途径，除此之外，还可以通过代码测试来提高代码的健壮性。常见的代码测试方式有：本地测试、Junit 测试、用例测试等。

2.5.1 使用Flow进行类型检查

使用 Flow 工具进行 React Native 应用开发不是必须需要掌握的技能，但是掌握了 Flow，可以帮助开发者开发出高质量的应用。

Flow 是一个静态的类型检查工具，设计之初就是为了可以发现 JavaScript 脚本里不容易被发现的错误。它依赖类型推断来检查类型错误，甚至可以检测注释代码，利用 Flow 提供的功能，开发者可以在开发过程中就发现问题从而避免在线上环境出现类似问题。

Flow安装与配置

Flow 的安装和配置都比较简单，在项目目录下使用如下命令：

```
npm install --save-dev flow-bin
```

然后我们使用命令运行 Flow，系统会为我们自动创建一个 .flowconfig 文件，它配置了 Flow 的行为，命令如下：

```
flow check
```

如果读者在项目中没有发现 flowconfig 文件，也可以手动新建一个 .flowconfig 文件，然后在 .flowconfig 文件中添加如下内容：

```
./node_modules/.*
```


然后，使用命令再次运行就可以正确地运行程序了。

2.5.2 Jest单元测试

Jest 是一款基于 Jasmine 的单元测试框架，它提供了侵入式的依赖自动模拟功能，也可以很好地与 React 测试工具进行整合。Jest 是系统默认为我们集成的代码测试框架，如果你的项目比较老，需要手动配置，过程如下。

Jest安装与配置

安装命令如下：

```
npm install jest-cli --save-dev
```

然后打开 package.json 文件，在 scripts 脚本片段中添加 test：

```
"scripts": {
  ...
  "test": "jest"
}
```

在 React Native 源代码的根目录中使用命令 `npm test` 来运行 jest 测试代码，测试代码会放置在 `_tests_` 目录下。默认情况下，系统初始化项目的时候，会在 `_tests_` 文件夹下创建 `index.ios.js` 和 `index.android.js` 两个测试文件专门用来编写测试配置。

当然也可以自己创建测试文件，关于使用 Jest 进行 Javascript 进行测试的更多技术细节，读者可以访问 Jest 的官网：<http://facebook.github.io/jest/>。

单元测试（Android）

单元测试直接在本地运行，不需要模拟器，React Native 使用 Buck 编译工具来运行测试用例，使用 Buck 的命令如下：

```
cd react-native
./scripts/run-android-local-unit-tests.sh
```

2.5.3 集成测试

集成测试，又称为组装测试，就是将软件产品中各个模块组装起来，检查其接口是否存在问题，以及组装后的整体功能、性能表现。在开展集成测试之前，往往要先进行深入的单元测试。

集成测试（Android）

集成测试运行在模拟器 / 真机上，以验证模块、组件以及 React Native 的内核部分（比如 bridge）在端对端测试中是否可以正常运作。在集成测试之前，确保正确安装和配置了 Android NDK 环境。React Native 使用 Buck 编译工具来运行测试，运行命令如下：

```
npm install
./scripts/run-android-local-integration-tests.sh
```

集成测试 (iOS)

React Native 提供了一些工具来简化跨原生与 JS 端的组件的集成测试, 这套工具的两个主要部分是 RCTTestRunner 与 RCTTestModule。RCTTestRunner 预设了 React Native 的环境, 并且以 XCTestCase 的形式在 Xcode 中直接运行。而 RCTTestModule 则以 NativeModules.TestModule 对象导出到 JS 环境中。测试代码需要以 JS 写成, 且必须在测试完成后调用 TestModule.markTestCompleted() 方法, 否则测试过程会超时并且失败。失败的表现一般是抛出一个 JS 异常。

Xcode 中运行 IntegrationTest 和 UIExplorer 两个官方示例时, 可以使用快捷键 cmd+U 来直接在本地运行集成测试。

快照测试 (iOS)

快照测试是集成测试的一种常见测试类型, 这类测试首先渲染一个组件, 然后使用 TestModule.verifySnapshot() 来比对屏幕截图与参考效果图的差别。参考效果图是通过在 RCTTestRunner 中设置 recordMode = YES, 然后在运行测试时录制的。屏幕截图在 32 位和 64 位色深以及不同的操作系统版本上可能会有细微的差别, 所以建议强制在指定的配置环境中执行快照测试。

在多人合作的项目开发中, 如果某人提交到仓库的代码会影响快照测试, 为了不影响原有的测试, 那么只需在 UIExplorer/UIExplorerSnapshotTests.m 中设置 _runner.recordMode = YES;, 然后重新运行先前失败的测试代码即可。

2.6 小结

本章主要从 React Native 开发环境的搭建、项目运行、项目调试、项目结构, 本地代码测试等方面介绍了 React Native 开发中比较实用的知识和开发技巧, 为后面的实战项目的开发打下基础。

下一章将从 React Native 开发的基础知识着手, 一步步深入介绍 React Native 开发常见的知识点, 进而一步步完成实战项目的开发。

第

3 章

React Native 开发基础

React Native 是一个专注 UI 构建的框架，设计的初衷是为了屏蔽平台的差异化显示，所以，界面是 React Native 开发中一个重要的组成部分。在本章中，主要从页面布局、JSX 基础语法、样式表等方面来介绍相关基础知识。

3.1 FlexBox 布局

FlexBox 是 flexible box 的缩写，译为“弹性盒布局”，FlexBox 是 React Native 开发中必不可少的页面组成部分，下面针对 FlexBox 的基础属性和实例做简单的讲解。

3.1.1 FlexBox 简介

做过前端开发的人都知道，传统的页面布局基于盒子模型而设计，主要依赖于定位属性、流动属性和显示属性。而处理一些伸缩性的布局，传统的盒子模型处理起来很麻烦。于是在 2009 年，由 W3C 组织提出了一种全新的布局方案，即 Flex 布局方案。该布局可以解决传统页面无法伸缩的问题，相比传统的盒子布局，FlexBox 更加灵活。

引入 FlexBox 布局模型的目的是提供一种更加有效的方式来对一个容器中的条目进行排列、对齐和空白空间分配。即便容器中条目的尺寸未知或是动态变化的，FlexBox 布局模型也能正常工作。在 FlexBox 布局模型中，容器会根据布局的需要，调整其中包含的条目的尺寸和顺序来恰当地填充所有可用的空间，当容器的尺寸由于屏幕大小或窗口尺寸发生变化时，其中包含

的条目也会动态地调整。

在 FlexBox 的官网: <http://caniuse.com> 中, 读者可以查看目前市场上大部分浏览器对 FlexBox 的支持情况, 如图 3-1 所示。

Current aligned Usage relative Date relative Show all									
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
			55			9.3		4.4	
		51	56	10	43	10.2		4.4.4	
41	11	52	57	10.1	44	10.3	all	56	57
	15	53	58	TP	45				
		54	59		46				
		55	60						

图3-1 浏览器对FlexBox的支持情况

FlexBox 主要由伸缩容器和伸缩项目组成, [flex-container] 定义一个伸缩容器, 伸缩容器中的每一个子元素都是一个伸缩项目。简单来说, Flexbox 就是为了伸缩容器内伸缩项目的布局而设计的。

3.1.2 FlexBox布局模型

FlexBox 布局的容器 (flex container) 指采用了 FlexBox 布局的 DOM 元素, 而 FlexBox 布局的条目 (flex item) 指容器中包含的子 DOM 元素, 如图 3-2 所示。

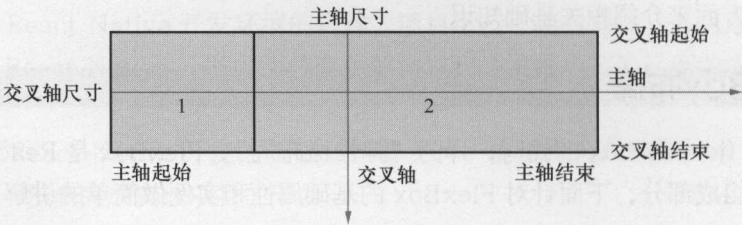


图3-2 FlexBox布局模型

由上图可知, FlexBox 布局主要由两个互相垂直的坐标轴组成: 主轴 (main axis) 和交叉轴 (cross axis)。主轴并不一定为水平方向的 x 轴, 交叉轴也不一定为垂直方向的 y 轴。在使用时, 通过 CSS 属性声明首先定义主轴的方向, 则交叉轴的方向也相应确定下来。容器中的条目可以排列成单行或多行。主轴确定了容器中每一行上条目的排列方向, 而交叉轴则确定行本身的排列方向。在实际应用中, 可以根据不同的页面设计要求来确定合适的主轴方向。

确定主轴和交叉轴的方向之后, 还需要确定它们各自的排列方向。对于水平方向上的轴, 可以从左到右或从右到左来排列; 对于垂直方向上的轴, 则可以从上到下或从下到上来排列。

对于主轴来说，排列条目的起始和结束位置分别称为主轴起始（main start）和主轴结束（main end），对于交叉轴来说，排列行的起始和结束位置分别称为交叉轴起始（cross start）和交叉轴结束（cross end）。在容器进行布局时，在每一行中会把其中的条目从主轴起始位置开始，依次排列到主轴结束位置；而当容器中存在多行时，会把每一行从交叉轴起始位置开始，依次排列到交叉轴结束位置。

FlexBox 布局中的条目有两个尺寸：主轴尺寸和交叉轴尺寸，分别对应其 DOM 元素在主轴和交叉轴上的大小。如果主轴是水平方向，则主轴尺寸和交叉轴尺寸分别对应于 DOM 元素的宽度和高度；如果主轴是垂直方向，则主轴尺寸和交叉轴尺寸要反过来。

3.1.3 FlexBox布局属性

FlexBox 布局的常用属性有：

- display-flex
- flex-direction
- flex-wrap
- flex-flow
- justify-content
- align-items
- align-content

display-flex

本属性用来指定元素是否使用弹性盒布局。当使用 FlexBox 布局的时候，需要先给父容器的 display 值定位 flex（块级）或者 inline-flex（行内级）。其语法为：

```
display: flex | inline-flex
```

下面简要介绍下这两个属性的含义。

flex 用于块级伸缩属性。示例代码如下：

```
.flex-container {
  display: flex;
  ...
}
```

inline-flex 用于行内伸缩属性。示例代码如下：

```
.flex-container {
  display: inline-flex;
  ...
}
```

flex-direction

本属性用来控制伸缩容器中主轴的方向，同时也决定了伸缩项目的方向。常用的属性

如下所示。

- flex-direction:row: 默认值，主轴的方向和正常的方向一致，从左到右排列。
- flex-direction:row-reverse: 和row的方向相反，从右到左排列。
- flex-direction:column: 从上到下排列。
- flex-direction:column-reverse: 从下到上排列。

示例代码如下：

```
.flex-container {  
  flex-direction: row;  
  ...  
}
```

flex-direction 属性常见展示效果如图 3-3 所示。

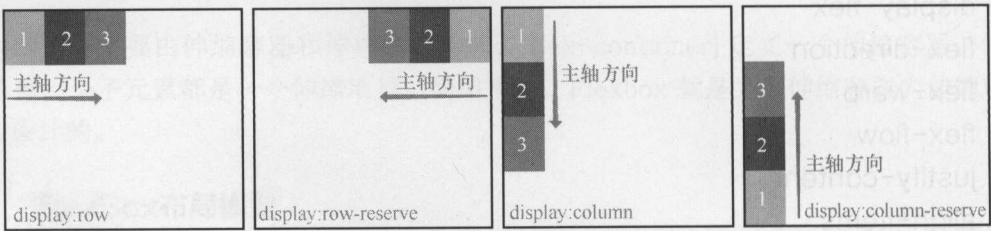


图3-3 flex-direction属性效果

flex-wrap

本属性用来控制伸缩容器是单行还是多行，也决定侧轴方向。常见的属性值如下所示。

- flex-wrap:nowrap: 默认值，伸缩容器单行显示。
- flex-wrap:wrap: 伸缩容器多行显示；排列顺序由上到下排列。
- flex-wrap:wrap-reverse: 伸缩容器多行显示，排列顺序由下到上排列。

flex-wrap 属性常见展示效果如图 3-4 所示。

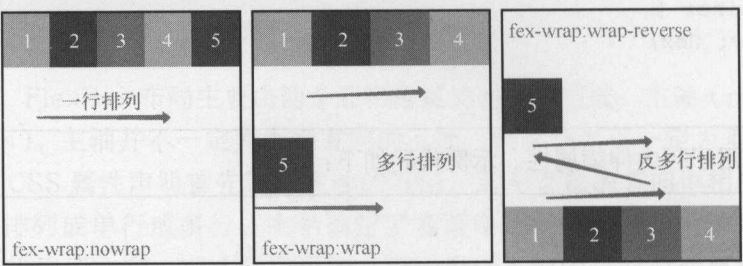


图3-4 flex-wrap属性效果

flex-flow

本属性是 flex-direction（主轴方向）和 flex-wrap（侧轴方向）的缩写。这两个属性决

定了伸缩容器的主轴与侧轴，默认值为 row nowrap。其语法为：

```
flex-flow: [flex-direction] [flex-wrap];
```

示例代码如下：

```
flex-flow: row wrap;
```

flex-flow 属性常见展示效果如图 3-5 所示。

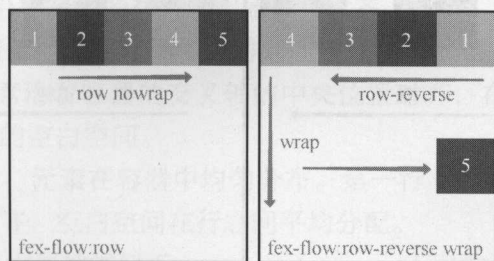


图3-5 flex-flow属性效果

justify-content

本属性用来定义伸缩项目在主轴上的对齐方式。当一行上的所有伸缩项目都不能伸缩或可伸缩但是已经达到其最大长度时，本属性会对多余的空间进行分配。

其常见的属性如下所示。

- justify-content: flex-start: 伸缩项目以主轴的起始位置开始对齐，后面的每个元素紧挨着前一个元素对齐。
- justify-content: flex-end: 伸缩项目以主轴的结束位置对齐，前面的每个元素紧挨着后一个元素对齐。
- justify-content: center: 伸缩项目相互对齐并在主轴上处于居中，并且第一个元素到主轴起点的距离等于最后一个元素到主轴终点的位置。
- justify-content: space-between: 伸缩项目平均的分配在主轴上，第一个元素和主轴的起点紧挨，最后一个元素和主轴上终点紧挨，中间剩下的伸缩项目进行平分。
- justify-content: space-around: 伸缩项目平均的分布在主轴上，第一个元素到主轴起点距离和最后一个元素到主轴终点的距离相等，且等于中间元素两两的间距的一半，完美地平均分配。

示例代码如下：

```
<View
  style={ {justifyContent:'center',flexDirection:'row',backgroundColor:"darkgray",
marginTop:20}} >
  </View>
```

just-content 属性常见展示效果如图 3-6 所示。

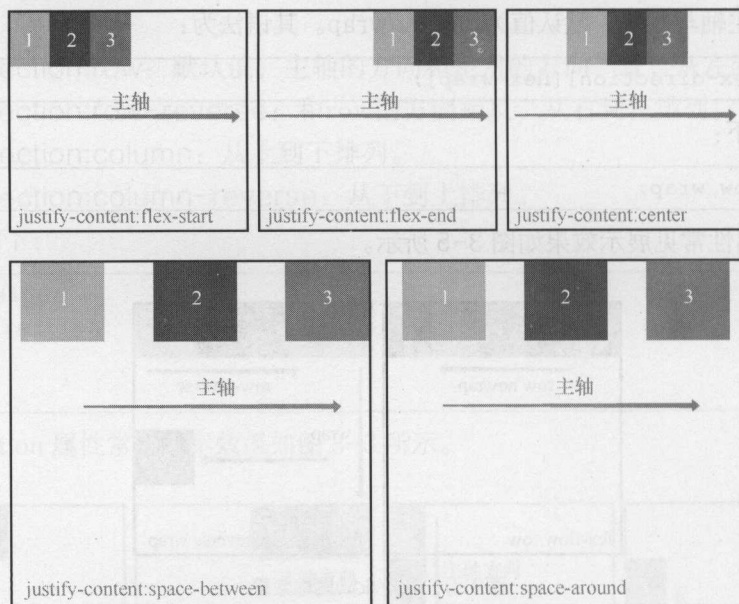


图3-6 justify-content属性效果

align-items

本属性用来定义伸缩项目在侧轴的对齐方式。flex-direction 和 flex-wrap 是一对，justify-content 和 align-items 是一对。前者分别定义主轴和侧轴的方向，后者分别定义主轴和侧轴中项目的对齐方式。其常见的属性如下所示。

- align-items: flex-start: 伸缩项目在侧轴起点边的外边距紧靠该行在侧轴起点的边。
- align-items: flex-end: 伸缩项目在侧轴终点边的外边距紧靠该行在侧轴终点的边。
- align-items: center: 伸缩项目的外边距在侧轴上居中放置。
- align-items: baseline: 如果伸缩项目的行内轴与侧轴为同一条，则该值与[flex-start]等效。
- align-items: stretch: 伸缩项目拉伸填充整个伸缩容器。

align-items 属性常见效果如图 3-7 所示。

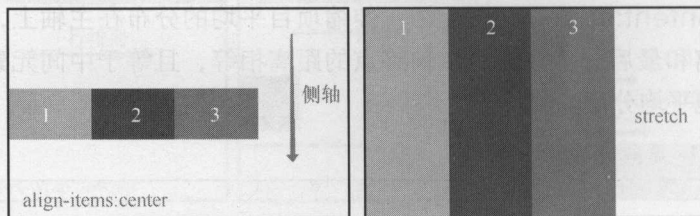


图3-7 align-items属性效果

align-content

本属性用来调整伸缩行在伸缩容器里的对齐方式。其作用类似于 justify-content，只不过

justify-content 表示的是主轴方向上对齐行中的条目。本属性在只有一行的伸缩容器上没有效果。

其语法为：

```
align-content: flex-start || flex-end || center || space-between ||  
space-around || stretch;
```

常见的属性如下所示。

- flex-start：元素沿着容器的交叉轴起始位置对齐。
- flex-end：元素沿着容器的交叉轴的结束位置对齐，和flex-start相反。
- flex-center：元素沿着容器的交叉轴的中央位置对齐，在交叉轴起始方向和结束方向上留有同样大小的空白空间。
- space-between：元素在容器中均匀分布。第一行与起始位置保持对齐，最后一行与结束位置保持对齐，空白空间在行之间平均分配。
- space-around：本属性类似于space-between，不同的是第一行条目和最后一个行目与容器行的边界之间同样存在空白空间，而空白空间的尺寸是行目之间的空白空间的尺寸的一半。
- stretch：元素被拉伸来占满剩余的空间。

align-content 属性常见的展示效果如图 3-8 所示。

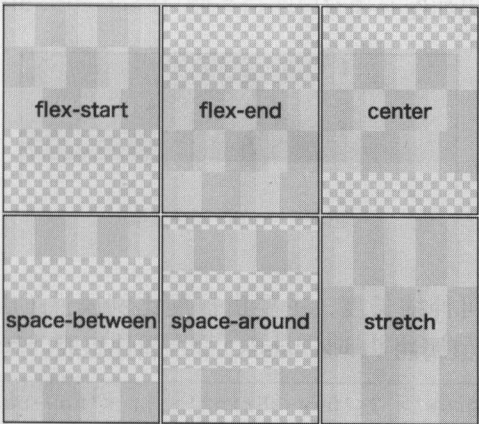


图3-8 align-content属性

3.1.4 FlexBox伸缩项目属性

伸缩项目属性主要包含如下 3 个子属性。

- order
- flex(flex-grow、flex-shrink、flex-basis)
- align-self

order

本属性主要用来定义项目的排列顺序，数值越小越靠前，默认值为 0。

例如，一个 container 中有 4 个 box，我们想让 box4 第一个显示，box1 为最后一个显示。那么我们只需要将 box4 设置为 [order:-1;] 既可。代码示例如下：

```
<style>
.container{
  display: flex;
}
.box1{
  order:1;
}
.box4{
  order:-1;
}
</style>
<div class="container">
  <div class="box1">1</div>
  <div class="box2">2</div>
  <div class="box3">3</div>
  <div class="box4">4</div>
</div>
```

order 属性代码运行效果如图 3-9 所示。

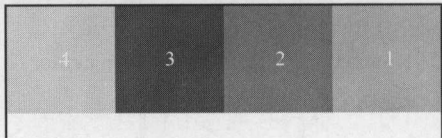


图3-9 order属性效果

flex

本属性主要用来指定可伸缩长度，主要由扩展比例属性（flex-grow）、收缩比例属性（flow-shrink）和伸缩基准值属性（flex-basis）3 个属性组成。其语法为：

```
flex:none | [ <'flex-grow'> ?<'flew-shrink'> || <'flow-basis'>]
```

- flex-basis: 本属性用来定义伸缩项目的基准值。默认是auto，即根据可伸缩比例计算出剩余空间的分布之前，伸缩项目主轴长度为起始数值。其语法为：

```
flex- basis:length|auto
```

- flex-grow: 本属性用来定义伸缩项目的放大比例，即用来决定伸缩容器剩余空间按比例应扩展多少空间。

默认值为 0，即如果存在剩余空间，也不放大。如果 flex-grow 设置为 1，即将每一个伸缩空间设置为一个大小相等的剩余空间。

- `flex-shrink`: 本属性用来定义伸缩项目的缩放比例, 即用来决定伸缩容器里其他伸缩项目的收缩空间比例。在收缩的时候收缩比例会以`[flex-basis]`伸缩基准值加权。

`align-self`

本属性用来覆写设置单独的伸缩项目在交叉轴上的对齐方式。该属性是用来覆盖伸缩容器属性 `align-items` 对每一行的对齐方式。也就是说, 在默认的情况下这两个值是相等的。其语法为:

```
align-self: auto | flex-start | flex-end | center | baseline | stretch
```

示例代码如下:

```
.flex-container {
  display: flex;
  ...
}
.item{
  align-self:auto
}
```

3.1.5 FlexBox在React Native中的应用

React Native 通过引入 FlexBox 的布局引擎, 可以在所有移动平台上实现了一致的跨平台样式和布局方案。

在 React Native 中, FlexBox 布局目前支持的属性有如下 6 个:

- `flex`
- `flexDirection`
- `alignSelf`
- `alignItems`
- `justifyContent`
- `flexWrap`

`flex`

本属性与 CSS 中的 `flex` 一样, 用来指定元素是否使用弹性盒布局。不过需要注意的是, 属性值大于 0 才可伸缩。其语法如下:

```
flex: number
```

`flexDirection`

本属性与 CSS 中的 `flex-direction` 一样, 用来指定元素的伸缩方向。`row` 表示横向伸缩, `column` 表示纵向伸缩。其语法如下:

```
flexDirection: row | column
```

`alignSelf`

本属性与 CSS 中的 `align-self` 一样, 用来设置单独的伸缩项目在交叉轴上的对齐方式。其

语法如下：

```
flexSelf: flex-start | flex-end | center | auto | stretch
```

alignItems

本属性与 CSS 中的 align-items 一样，用来设置伸缩项目在侧轴的对齐方式。其语法如下：

```
flexItems: flex-start | flex-end | center | stretch
```

flexWrap

本属性与 CSS 中的 flex-wrap 一样，用来控制伸缩项目是单行还是多行展示。其语法如下：

```
flexWrap: wrap | nowrap
```

justifyContent

本属性与 CSS 中的 justify-content 一样，用来控制伸缩项目在主轴上的对齐方式。其语法如下：

```
justifyContent: flex-start | flex-end | center | space-between | space-around
```

3.1.6 FlexBox综合实例

下面我们将通过示例程序来加强对 FlexBox 布局的理解。其最终的界面如图 3-10 所示。



图3-10 FlexBox综合实例

图 3-10 的界面主要由两部分组成，HTML 和 CSS 样式文件，而在排版的时候使用 FlexBox 替换百分比布局。

HTML 核心代码如下：


```

<div class="main">
  <div class="header common">
    <div class="logo"></div>
    <div class="title">Flexbox</div>
  </div>

  <div class="main-center">
    <div class="item left common">
      <a href="#">专栏</a>
    </div>
    <div class="item right common">
      <a href="#">通告</a>
    </div>
  </div>

  <div class="footer common">
    <span>Copyright © <strong>xiangzhihong</strong>
    All Right Reserved</span>
  </div>
</div>

```

CSS 样式文件代码如下:

```

body {
  background-color: #333;
  font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
  font-size: 100%;
}

.footer {
  position: relative;
  width: 100%;
  height: 160px;
  background-color: #345AA3;
  color: #fff;
  font-size: 100%;
  text-align: center;
  -Webkit-flex-direction: column;
  flex-direction: column;
}

.main-center .left,
.main-center .right {
  -Webkit-flex-direction: column;
  flex-direction: column;
}

```

```
.main-center img {
  width: 80px;
}

.footer {
  background-color: #999;
  color: #666;
  font-size: 100%;
}

.main-center,
.common {
  display: -Webkit-flex;
  display: -moz-flex;
  display: -ms-flex;
  display: flex;
  -Webkit-justify-content: center;
  justify-content: center;
  -Webkit-align-items: center;
  align-items: center;
}

.item {
  width: 48%;
  margin: 1% 0;
  color: #fff;
  min-height: 160px;
}

.left {
  background-color: #913E8E;
}

.right {
  background-color: #00B1BD;
}

.userface {
  width: 120px;
}
```

3.2 ES6语法基础

ECMAScript 6 (以下简称 ES6) 是 JavaScript 语言的下一代标准, 因为 ES6 是在 2015 年发布的, 所以又称其为 ECMAScript 2015。和上一版 ES5 不同, ES6 引入了大量的新思路语法规则, 本节重点讨论 ES6 常用的一些新特性。

目前，并不是所有浏览器都兼容 ES6，但是主流的浏览器都支持 ES6，并且服务器端的代码基本上都支持 ES6 语法，越来越多的 IDE 纷纷默认支持 ES6 语法。对于一些老的项目，必要的时候还可以使用 Babel 将 ES6 转为 ES5。

ES6 提供了许多新的语法和代码特性来提高 JavaScript 的性能，并且在代码结构上更加简洁易懂。下面列举一些常见的特性。

3.2.1 组件的导入与导出

组件导出

在组件导出方面，在 ES5 语法里，导出类通过 `module.exports` 来导出。

```
//ES5 导出类
var MyComponent = React.createClass({
  ...
});
module.exports = MyComponent;
```

在 ES6 语法里，使用 `export default` 来导出类或组件。

```
//ES6 导出类
export default class MyComponent extends Component{
  ...
}
```

组件导入

在组件的导入方面，在 ES5 语法里，使用 CommonJS 标准导包，通过 `require` 来导入组件。例如：

```
//ES5 导入 React Native 组件
var ReactNative = require("react-native");
var {
  Image,
  Text,
} = ReactNative;
//ES5 导入自定义组件
var MyComponent = require('./MyComponent');
```

在 ES6 语法里，则使用 `import` 方式导入组件。

```
//ES6 导入 React Native 组件
import {
  Image,
  Text
} from 'react-native'
// ES6 导入自定义组件
import MyComponent from './MyComponent';
```

3.2.2 类

在传统的 JavaScript 中，通常使用 function 和 prototype 来模拟类的概念，每个函数（function）就是一个对象，而每个函数对象又包含一个 prototype 对象。但是，在 ES6 中引入了 class 关键字，添加了对类的支持（其实，在 JavaScript 中 class 一直作为保留字使用），使用类的概念之后，对象的创建和继承更加直观，对父类的调用、实例化、静态方法和构造函数等概念都更加形象化，代码结构也更加清楚直观。JavaScript 本身就是面向对象的语言，ES6 中提供的类，实际上更方便了开发者对 JavaScript 原型模式的包装。

创建类

在 ES5 里，通过 React.createClass 来创建一个组件类。

```
//ES5 创建类
var Demo = React.createClass({
  ...
});
```

在 ES6 里，通过继承自 React.Component 的 class 来定义一个组件类。

```
//ES6 创建类
class Demo extends React.Component {
  ...
}
```

组件类方法

在传统的 JavaScript 中，通常使用 function 函数来给组件定义方法。在 ES6 中，则可以直接使用函数名字来定义方法，在方法结束外也不需要使⽤逗号。

```
//ES5 定义类内部方法
var Demo = React.createClass({
  componentWillMount: function() {
    },
  ...
});
//ES6 定义类内部方法
var Demo = React.createClass({
  componentWillMount() {
    }
  ...
});
```

组件的属性与属性类型

在 ES5 语法中，属性类型和默认属性通过 propTypes 和 getDefaultProps() 来实现。例如：

```
//ES5 属性与属性类型定义
var Video = React.createClass({
```

```

getDefaultProps: function() {
  return {
    autoPlay: false,
    maxLoops: 10,
  };
},
propTypes: {
  autoPlay: React.PropTypes.bool.isRequired,
  maxLoops: React.PropTypes.number.isRequired,
  videoSrc: React.PropTypes.string.isRequired,
},
...
});

```

在 ES6 里，统一使用 `static` 成员来修饰属性类型和默认属性。例如：

```

//ES6 属性与属性类型定义
class Video extends React.Component {
  static defaultProps = {
    autoPlay: false,
    maxLoops: 10,
  };
  static propTypes = {
    autoPlay: React.PropTypes.bool.isRequired,
    maxLoops: React.PropTypes.number.isRequired,
    videoSrc: React.PropTypes.string.isRequired,
  };
  ...
}

```

3.2.3 状态变量

在 React 框架中，所有的界面被视为一个简单的状态机，那么任意一个 UI 场景就是状态机中的一种状态。状态机的状态一旦发生变化，就会触发界面的重新渲染。

在 ES5 语法中，React Native 的状态变量在 `getInitialState()` 中声明。例如：

```

//ES5 状态机变量声明
let Index = React.createClass({
  getInitialState:function(){
    return {
      var1:'value of var1',
      var2:30,
      var3:true
    };
  });
});

```

而在 ES6 语法中，状态机变量的声明必须在组件的构造函数中声明。例如：

```
//ES6 状态机变量声明
var Index extends Component {
  constructor(props) {
    super(props);
    this.state = {
      var1: 'value of var1',
      var2:30,
      var3:true
    };
  }
}
```

3.2.4 回调函数

在 ES5 语法中，回调函数可以直接调用本组件的某个成员方法。例如：

```
//ES5 添加回调函数
let Login = React.createClass({
  getInitialState:function () {
    return {
      inputedNum:''
    };
  },
  updateNum: function (num) {
    // 更新成员数量作为回调函数
    this.setState((state) => {
      return {
        inputedNum:num
      };
    });
  },
  // 省略...
  render:function () {
    return(
      <View style={styles.container}>
        <TextInput style={styles.innerViewStyle}
          placeholder={"请输入账号"}
          // 指定第一个回调函数
          onChangeText={(num) => this.updateNum(num)}
        />
        // 省略...
      </View>
    );
  }
});
```

在使用 ES6 语法编写 React Native 组件时，组件的回调函数必须在组件的构造函数中执行绑定操作。使用 ES5 语法绑定组件操作也是在这一步，但是 React 类的 `createClass()` 代劳了这一操作，`React.createClass()` 会把所有的方法都绑定一遍。使用 ES6 绑定回调函数时，开发者必须手动在代码中添加绑定操作。例如：

```
//ES6 开发者在构造函数中手动添加绑定
class Login extends Component {
  constructor (props){
    super(props);
    this.state = {
      inputedNum:''
    };
    // 手动绑定回调函数
    this.updateNum = this.updateNum.bind(this);
  }
  updateNum(num) {
    this.setState((state) => {
      return {
        inputedNum: num
      };
    });
  }
  // 省略...
  render() {
    return (
      <View style={styles.container}>
        <TextInput style={styles.innerViewStyle}
          placeholder={"请输入账号"}
          // 指定回调函数
          onChangeText={( num) => this.updateNum(num)}
        />
      </View>
    );
  }
}
```

相对于 ES5 语法，使用 ES6 语法开发需要开发者自己手动绑定每一个回调函数，这对于开发者来说似乎是一种开发方便性上的退步。

3.2.5 参数

ES6 对参数的写法做了较大的改动，主要体现在参数默认值、不定参数、拓展参数方面。

参数默认值

通常来说，函数调用者不需要传递所有可能存在的参数，没有被传递的参数可由默认的参数进行填充。JavaScript 有严格的默认参数格式，未被传值的参数默认为 `undefined`。

在 ES5 语法中，对于默认值的设定，往往需要通过逻辑或操作符运算来设置。而在 ES6 中，系统允许开发者在函数定义的时候就指定任意参数的默认值。例如：

```
// 设置默认参数
function sayHello(name){
    // 传统的指定默认参数的方式
    var name=name||'dude';
    console.log('Hello '+name);
}
// 运用 ES6 的默认参数
function sayHello2(name='dude'){
    console.log('Hello ${name}');
}
sayHello();           // 输出:Hello dude
sayHello('Wayou');    // 输出:Hello Wayou
sayHello2();          // 输出:Hello dude
sayHello2('Wayou');   // 输出:Hello Wayou
```

不定参数

在传统的 JavaScript 代码中开发者可以通过 arguments 变量来接受任意数量的参数，例如：string.prototype.concat() 就可以接受任意数量的字符串参数。ES6 提供了一种可变参函数的新方式——不定参数。

不定参数指在函数中使用命名参数，同时接收不定数量的未命名参数。不定参数的格式是 3 个句点后跟代表所有不定参数的变量名。例如：

```
// 将所有参数相加
function add(...x){
    return x.reduce((m,n)=>m+n);
}
// 传递任意个数的参数
console.log(add(1,2,3)); // 输出结果: 6
```

拓展参数

ES6 语法提供了一种新的参数格式，它允许传递数组或者类数组，直接做为函数的参数，而不用通过 apply 方法。例如：

```
//ES6 扩展参数
var people=['Wayou','John','Sherlock'];
function sayHello(people1,people2,people3){
    console.log('Hello ${people1},${people2},${people3}');
}
// 以拓展参数的形式传递
sayHello(...people);           // 输出:Hello Wayou,John,Sherlock
//ES5 语法格式，需要使用函数的 apply 方法
sayHello.apply(null,people);    // 输出:Hello Wayou,John,Sherlock
```

3.2.6 箭头操作符

ES6 新增了一种新的语法格式：箭头操作符 (\Rightarrow)，用来简化函数的书写。该操作符左边为输入参数，使用圆括号包含参数部分，而右边则是操作结果或返回的值。例如，在 JavaScript 中，回调是一种常见的操作，在 ES6 之前，每次使用匿名回调函数都需要写一个 function，甚是繁琐。

基本语法格式如下：

```
(形参列表) => { // 函数体 }
```

示例代码如下：

```
var array = [1, 2, 3];
// 传统写法
array.forEach(function(v, i, a) {
  console.log(v);
});
//ES6
array.forEach(v => console.log(v));
```

3.2.7 Symbol

在 ES5 之前没办法创建私有变量，只能通过封装来解决，而 symbol 的到来为 JavaScript 开发者带来了福音。利用 Symbol 创建私有变量的代码如下：

```
// 创建一个 Symbol
let name= Symbol();
let person = {};
person[name] = "张三";
```

使用 Symbol 创建私有变量的时候，还可以传入字符串。例如：

```
var s1 = Symbol("abc");
var s2 = Symbol("abc");
console.log(s1 == s2); //false
```

说明：任意两个 Symbol 创建的变量都不相等，即使使用了相同的参数。

Symbol 作为基础类型，开发者可以使用 typeof 操作符来判断变量是否为 Symbol 类型，如果是 Symbol 类型则返回 “symbol”。例如：

```
let symbol = Symbol();
console.log(typeof symbol); // 输出结果 "symbol"
```

由于每个 Symbol 的值都是不相等的，这意味着 Symbol 的值可以作为标识符用于对象的属性名，从而保证不会出现同名的属性。例如，下面两种写法其结果是相同的：

```
var mySymbol = Symbol();
// 第一种写法
```

```

var a = {};
a[mySymbol] = 'Hello!';
// 第二种写法
var a = {
  [mySymbol]: 'Hello!'
}

```

Symbol属性名遍历

Symbol 作为属性名既不能作为 for 循环的对象，也不能使用 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 来操作 Symbol 属性。但是可以通过 Object.getOwnPropertySymbols() 获取指定对象的所有 Symbol 属性名。相关示例代码如下：

```

var obj = {};
var a = Symbol('a');
var b = Symbol('b');

obj[a] = 'Hello';
obj[b] = 'World';
// 返回 obj 对象所有 Symbol 类型的属性名组成的数组
var objectSymbols = Object.getOwnPropertySymbols(obj);
console.log(objectSymbols)
// 输出结果 [Symbol(a), Symbol(b)]

```

Symbol.for()和Symbol.keyFor()

Symbol.for (字符串参数)：在全局环境中搜索以该字符串作为参数 Symbol 值，如果搜到则返回这个 Symbol，如果搜不到则创建一个 Symbol，并把它注册在环境中。

```

var obj = {};
// 第一次搜不到，则新创建一个
var a = Symbol.for("tom");
// 第二次加入搜索
var b = Symbol.for("tom");
console.log(a == b); // 返回 true

```

Symbol.keyFor(symbol)：返回一个已经注册的 Symbol 对象的 key。

```

var a = Symbol("tom");
var b = Symbol.for("tom");
console.log(Symbol.keyFor(a)); // 输出 undefined
console.log(Symbol.keyFor(b)); // 输出 tom

```

3.2.8 解构

在 ES5 或更早的版本中，若一个函数要返回多个值，常规的做法是返回一个对象，将每个值作为这个对象的属性返回，这样就需要书写很多相似的代码。例如：

```
// 从对象中获取数据
let options = {
    repeat: true,
    save: false
};
let repeat = options.repeat,
    save = options.save;
```

而在 ES6 中，利用解构这一特性，可以直接返回一个数组或者对象，然后利用数组或对象的属性将值解析到对应的变量中。

对象解构

对象结构，就是在赋值语句的左侧使用类似对象的结构。例如：

```
let node = {
    type: " student ",
    name: "tom"
};
// 对象结构
let { type, name } = node;
```

如果你想要改变声明变量的值，可以使用解构表达式。例如：

```
let node = {
    type: "student",
    name: "tom"
},
type = "teacher",
name = "jack";
// 注意：必须要在圆括号内才能使用解构表达式
({type, name} = node)
```

在上面的操作中，都是将对象的属性值赋值给同名变量，其实还可以将属性值赋值给不同的名的变量，不过需要注意的是，冒号后面才是要定义的新的变量。例如：

```
let node = {
    type: "teacher"
};
let { type: localType, name: name = "tom" } = node;

console.log(localType); // "teacher"
console.log(name); // "tom"
```

数组解构

数据解构和对象解构类似，只是将操作的对象换成了数组，而解构操作使用的是数组内部的索引。例如：

```
let colors = [ "red", "green", "blue" ];
```

```
let [ firstColor, secondColor ] = colors;
console.log(firstColor);           // "red"
console.log(secondColor);         // "green"
```

如果只读取数组中某一项的值，对于其他的选项可以不用命名。例如：

```
let colors = [ "red", "green", "blue" ];
// 只取数组中的第三项的值
let [ , , thirdColor ] = colors;
console.log(thirdColor);           // "blue"
```

在数组解构中有一个常用的功能是交换两个变量的值，在传统的写法中，我们需要使用第三方变量进行交换。例如：

```
let a = 3, b = 4, temp;
temp = a;
a = b;
b = temp;
console.log(a);
console.log(b)
```

而 ES6 完全抛弃了这种第三方变量的方式，直接使用数组解构表达式。

```
temp = a;
let a = 3, b = 4;
// 利用数组解构表达式交换变量值
[a, b] = [b, a];
console.log(a);
console.log(b)
```

3.3 React JSX

React 使用 JSX 来替代常规的 JavaScript，然后通过工具（如 Babel）将 JSX 代码编译成 React 支持的 JS 文件，使用 JSX 可以让代码可读性更高、语义更清晰。React 主要由 ReactJS 和 React Native 构成，ReactJS 是 Facebook 开源的一个前端框架，React Native 是 ReactJS 原生开发的体现。

3.3.1 JSX入门

JSX 是 Facebook 团队提出的一个语法方案，可以在 JavaScript 代码中直接使用 HTML 标签来编写 JavaScript 对象。其使用的是 XML-like 语法，这种语法方案需要通过 JSXTransformer 进行编译，转换成真实可用的 JavaScript 代码。

React 的核心机制就包括虚拟 DOM，使用 JSX 语法可以很方便地创建虚拟 DOM。例如：

```
var root = (
  <ul className="list">
```

```

    <li>Content of node1</li>
    <li>Content of node2</li>
  </ul>
);

```

上面的代码等价于下面的 JavaScript 代码:

```

var node1 = React.createElement('li', null, 'Content of node1');
var node2 = React.createElement('li', null, 'Content of node2');
var rootNode = React.createElement('ul', { className: 'myList' }, node1, node2);

```

因为使用 JSX 可以让代码可读性更高、语义更清晰,并且 JSX 利用虚拟 DOM 技术减少对实际 DOM 的操作从而提升了性能。所以,使用 JSX 进行前端页面开发成为了行业标准之一。

3.3.2 JSX语法

在前端框架中,React 的开发思想是基于组件来开发产品,它将一个组件视为一个完全独立的、没有任何其他依赖的模块文件。React 发明了 JSX,利用特殊的语法格式来创建虚拟 DOM,而利用虚拟 DOM 可以减少对实际 DOM 的操作,从而提升性能。

JSX 语法和 XML 语法类似,可以定义属性以及子元素,唯一的区别是,JSX 用大括号来加入 JavaScript 表达式。JSX 必须借助 ReactJS 环境才能运行下面介绍 JSX 中常见的一些语法。

载入方式

目前,加载 JSX 文件主要有两种方式:内联方式载入和外联方式载入。内联方式载入的相关示例如下:

```

<script type="text/babel">
  ReactDOM.render(
    <h1>hello hangge.com</h1>,
    document.getElementById('example')
  );
</script>

```

外联方式,是将 JSX 代码单独放在一个 .JSX 文件中,然后再使用的文件中引入即可。外联方式载入的示例如下:

```

ReactDOM.render(
  <h1>hello hangge.com</h1>,
  document.getElementById('example')
);

```

然后,在其他文件引入之前定义的 JSX 文件,相关示例代码如下:

```

<script type="text/babel" src="hello.JSX"></script>

```

JSX 标签，其实就是 HTML 标签。例如：

```
<h1>Hello JSX</h1>
```

在 JavaScript 中书写这些标签时，不再需要使用引号将字符串引用起来，而是像直接书写 xml 文件一样即可。然而还有一类标签是 HTML 标签所没有的，就是 ReactJS 创建的组件类标签。ReactJS 创建的组件类标签其首字母必须大写。例如，创建一个自定义标签：

```
class Hello extends React.Component {
  render() {
    return (
      <div> hello </div>
    );
  }
}
```

三目表达式示例如下：

```
var person = <Person name={isLoggedIn ? name : ''} />;
```

- 数组递归：对数组进行循环，返回每个元素的 React 组件。例如：

```
var lis = this.todoList.todos.map(function (todo) {
  return (
    <li>
      <input type="checkbox" checked={todo.done}>
      <span className={'done-' + todo.done}>{todo.text}</span>
    </li>
  );
});
var ul = (
  <ul className="unstyled">
    {lis}
  </ul>
);
```

- 事件绑定：JSX 可以给元素直接绑定事件，如点击事件。React 并不会真正绑定事件到每一个具体的元素上，而是采用事件代理的方式，在根节点 document 上为每种事件添加唯一的事件监听者（Listener），然后通过事件的目标函数（target）找到真实的触发元素的相关事件。例如，

```
<button onClick={this.checkAndSubmit.bind(this)}>Submit</button>
```

- 属性：在 JSX 中可以通过标签的属性来改变当前元素的样式。例如：

```
var property = <h1 width="10px">Hello, React Native</h1>;
```

在 JSX 中，我们可以自定义属性，但是自定义属性必须以“data-”开头，这样才能渲染

到界面上。例如：

```
var hello = <h1 data-test="test" test="test"> Hello React Native</h1>
React.render(
  hello,
  ...
);
```

如上所示，data-test 标签能够被渲染到页面上，而 test 标签却不能。

- 样式：在前端Web开发中，我们会将样式文件写在独立的CSS文件中。在JSX中，如果功能相对单一，我们还可以将样式直接写在JSX中。例如：

```
<h1 style={{color: '#ff0000', fontSize: '15px'}}>Hello React Native</h1>
```

- 自定义组件：在JSX中，我们可以使用React自带的一些组件，也可以自定义组件。组件定义之后，可以利用XML语法去声明，而能够使用的XML Tag就是当前JavaScript上下文的变量名，该变量名即为组件名称。例如：

```
class HelloWorld extends React.Component{
  render() {
    return (
      <p>
        Hello, <input type="text" placeholder="Your name here" />!
        It is {this.props.date.toTimeString()}
      </p>
    );
  }
};

setInterval(function() {
  React.render(
    <HelloWorld date={new Date()} />,
    document.getElementById('example')
  );
}, 500);
```

上面声明了一个名为 HelloWorld 的组件，当需要使用的时候，先导入组件，然后直接在xml中直接使用即可。代码如下：

```
var MyHelloWorld = HelloWorld;
React.render(<MyHelloWorld />, ...);
```

当然也可以自定义命名空间，然后使用命名空间的方式引入。例如：

```
var sampleNameSpace = {
  MyHelloWorld: HelloWorld
```

```
};  
React.render(<sampleNameSpace.MyHelloWorld />, ...);
```

3.4 样式

前面说过, React Native 和 Web 应用程序除了逻辑代码不同之外, 样式部分是可以共享的。Github 上有一个 React Style 项目 (<https://github.com/JS-next/react-style>), 该项目提供 React Native 和 Web 样式的共用解决方案。在 React Native 的样式开发中, React Native 不实现 CSS, 而是依赖于 JavaScript 来为应用程序设置样式。

3.4.1 申明与操作样式

在使用 React 开发 web 应用的过程中, 为了让程序的结构更加清晰, 通常需要将逻辑代码和样式表分离出来, 而样式通常使用 CSS、SASS 编写。而在 React Native 的开发过程中, React Native 采用了完全不同的方式, 它将样式带入了到 JavaScript 的世界, 使用 JavaScript 对单个元素的样式进行操作即可。毋庸置疑, 这种方式摒弃了 CSS 的样式规范, 增加了代码的可读性。

为了更好地理解 React Native 样式的设计思想, 我们需要了解传统的 CSS 样式规范的痛点。传统的样式表最大的问题在于维护比较困难, 代码结构混乱。在传统的 CSS 样式规范中, CSS 代码通常都在全局作用域里, 如果不注意, 一个组件的样式很容易影响到其他组件。如 Twitter 公司很流行的 Bootstrap 库同时引入了 600 多个全局变量。由于 CSS 并非显示的 HTML 元素, 如果有些样式被抛弃不用, 想要消除这些无用的样式就会变得非常困难。

React Native 则采用了与传统的样式表截然不同的方式, 它将样式引入到 JavaScript 中, 从而保证了组件的模块化和独立化, 这也是 React Native 开发的优势之一。

语法

样式表由选择符和声明组成, 声明又由属性和值组成。简单来说, 声明就是英文大括号“{ }”, 属性和值之间用英文冒号“:”分隔, 多条申明用英文分号“;”分隔。例如:

```
Style{  
  font-size:12px;  
  color:red;  
}
```

3.4.2 样式分类

样式表按作用域分为: 行内样式、内嵌样式、外部样式。

行内样式

从语法上来讲, 行内样式是编写组件样式中最简单的方式, 但代码不利于查看和维护。行

内样式的写法就是将 CSS 代码直接写在标签中，直接设置元素的 style 属性。如果有多条 CSS 样式，也可以写在一起，中间用分号隔开。例如：

```
<View style={
  {
    width: 300,
    height: 600,
    backgroundColor: 'red',
  }
}>
```

内嵌样式

内嵌样式又称为对象样式，是对行内样式的一种升级写法，把 CSS 代码写在该文件，而不是将样式直接写在标签中。例如：

```
var bold={
  fontWeight:'bold'
};
...
render(){
  return(
    <Text style={bold}>
      内嵌样式
    </Text>
  )
}
```

外部样式

外部样式又称为外联样式，是指将 CSS 代码写在一个单独的外部文件中，然后在使用的地方导入样式文件。例如 style.js：

```
import React from 'react';
import {
  StyleSheet
} from 'react-native';

var style= StyleSheet.create(
{
  fontSize: {
    fontSize: 20
  }
});

module.exports = style;
```

在使用样式的时候，先导入样式文件，然后再使用。例如：

```
import style from './styles'
...
<Text style={style.fontSize}>
  外联样式
</Text>
```

3.4.3 样式使用

StyleSheet.create

在 React Native 官方的示例代码中，很多地方都使用 `StyleSheet.create` 来实例化样式对象。使用 `StyleSheet.create` 来创建样式有很多的优势。`StyleSheet` 通过将样式文件转换为一个引用内部表的纯数字，来确保值是不可变和不透明的。通过将 `StyleSheet` 放在样式文件的最后，也确保了样式文件为只被创建一次，而不是每一个渲染周期都被创建。

换言之，`StyleSheet.create` 是提供保护的语法糖，它提供了通过 `propTypes` 校验属性的能力。在样式的使用过程中，除了上面介绍的几种使用方式之外，开发者还可以对样式进行复用和拼接操作。

`style` 属性接受对象数组：例如，使用 `StyleSheet.create` 创建一个样式数组的代码如下：

```
var styles = StyleSheet.create({
  button: {
    backgroundColor: '#ffffff',
    borderRadius: 8
  },
  text: {
    fontSize: 20
  }
});
```

然后，通过简单的样式拼接操作即可创建一个拥有组合样式的组件。

```
var combination=React.createClass({
  render() {
    return (
      <Text style={[styles.button,styles.text]}>
        { 拼接组件 }
      </Text>
    );
  }
});
```

React Native 的属性不仅可以接受对象数组，还可以在对象数组的基础上接受内联样式，

不过这样写出来的代码有点杂乱无章，阅读性差。例如：

```
var combination=React.createClass({
  render() {
    return (
      <Text style={[styles.button,styles.text,{color:'#000000'}]}>
        { 内联样式拼接组件 }
      </Text>
    );
  }
});
```

值得注意的是，在使用数组对象样式时，如果有两个对象的样式属性相同，React Native 会默认选择最后一个，并且对于空值如：（false、null 和 undefind）会被忽略。

除了上面的特性之外，React Native 还支持条件性样式，只需要在 Style 中添加额外的条件语句即可。例如：

```
<View style={this.state.touching&&styles.highlight}/>
```

对比一下 Web 样式的做法，我们采用 @extend 关键字，或者在 CSS 中对类进行嵌套和重写。虽然使用拼接样式可以到达预期的效果，但是它是一种受限的写法，我们更多的是使用样式的导出和继承。不过拼接样式也有它的优点：它保持了逻辑的简洁性，让我们更容易查看组件的相关属性。

3.4.4 样式传递

为了能够在调用组件的时候对其子组件样式进行自定义，可以通过将样式作为参数进行传递。使用 View.propTypes.style 或 Text.propTypes.style 来确保传递的参数确实是 style 类型。

```
var List = React.createClass({
  propTypes: {
    style: View.propTypes.style,
    elementStyle: View.propTypes.style,
  },
  render: function() {
    return (
      <View style={this.props.style}>
        {elements.map((element) =>
          <View style={[styles.element, this.props.elementStyle]} />
        )}
      </View>
    );
  }
});
```

```
// ... in another file ...
<List style={styles.list} elementStyle={styles.listElement} />
```

总体来说，和传统概念的 CSS 样式相比，React Native 对传统的 CSS 规范进行了部分修改，并增加了一些新的规范以适应开发需求。

3.5 手势与触摸事件

传统的 Web 接口开发都是基于鼠标控制设计的，我们使用 hover 这样的状态来进行动态变换，并对状态的变换做出相应的响应处理。而对于移动设备而言，触控显得尤为重要。触控是移动设备的核心功能，也是移动应用交互的基础，Android 和 iOS 都有各自完善的触摸事件处理机制。而在 React Native 开发中，React Native 提供了一套统一的处理方式，能够方便地处理界面中组件的触摸事件、用户手势等。

3.5.1 触摸事件

React Native 提供的组件中除了 Text，其他组件默认是不支持点击事件的，也不能响应基本触摸事件，所以 React Native 提供了几个可以直接处理响应事件的组件，基本上能够满足大部分的点击处理需求。这些组件包括：TouchableHighlight、TouchableNativeFeedback、TouchableOpacity 和 TouchableWithoutFeedback。下面我们以 TouchableHighlight 为例进行介绍。

通常来讲，用户触摸任何界面元素响应结果都需要使用 <TouchableHighlight> 组件来包装，<TouchableHighlight> 组件提供的主要响应事件包括：onPressIn、onPressOut、onPress、onLongPress 等。例如，可以通过下面的代码来测试响应事件：

```
onPressIn() {
  console.log("press in");
}

onPressOut() {
  console.log("press out");
}

onPress() {
  console.log("press");
}

onLongPress() {
  console.log("long press");
}

render() {
  return (
```

```

<View style = {styles.container} >
  <TouchableHighlight
    style = {styles.touchable}
    onPressIn = {this._onPressIn}
    onPressOut = {this._onPressOut}
    onPress = {this._onPress}
    onLongPress = {this._onLonePress} >
    <View style = {styles.button} >
      </View>
    </TouchableHighlight>
  </View>
);
}

```

TouchableHighlight使用实例

下面是使用 TouchableHighlight 组件的 Touch 功能，实现点击文字从而弹出警告框的效果。如图 3-11 所示。

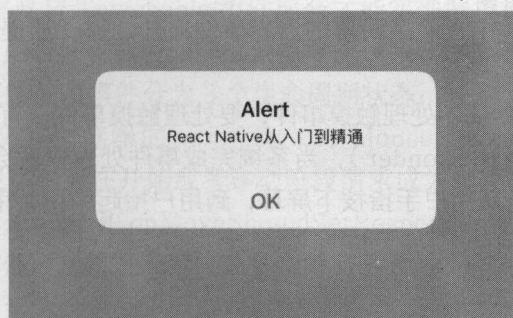


图3-11 TouchableHighlight点击弹窗

系统就会调用 TouchableHighlight 组件提供的 onPress 方法，将其放在段首位置，同时响应对应的事件，弹出警告框。

```

// 定义弹框
show() {
  alert('React Native 从入门到精通');
}
// 省略...
<TouchableHighlight
  style={styles.container}
  onPress={() => this.show()} >
  <Text style={styles.text}>React Native 从入门到精通 </Text>
</TouchableHighlight>

```

除了 TouchableHightLight 组件实现了 Touch 功能之外，其他 3 个组件同样也实现了 Touch 功能，它们的主要功能和区别如下。

TouchableHighlight

本组件用于封装视图，用来正确响应触摸操作。当用户按下按钮的时候，封装的视图的透明度不会降低，同时底层颜色显示出来，使得视图变暗或变亮。如果使用的方法不恰当，有时候会导致一些不希望出现的视觉效果。

TouchableNativeFeedback (仅限于Android)

本组件用于封装视图，用来正确响应触摸操作(仅限 Android 平台)。在 Android 设备上，这个组件利用原生状态来渲染触摸的反馈，目前它只支持一个单独的 View 实例作为子节点。

TouchableOpacity

本组件用于封装视图，用来正确响应触摸操作。当用户按下按钮的时候，封装的视图的不透明度会降低。这个过程并不会真正改变视图层级，大部分情况下能很容易地添加到应用中而不会带来一些奇怪的副作用。此组件与 TouchableHighlight 的区别在于不会有额外的颜色变化。

TouchableWithoutFeedback

本组件用来封装视图，但是不会处理回调信息，一般不会使用这个组件。

在移动设备的交互设计中，除了简单的点击事件之外，往往还提供了更加复杂的手势响应系统，如多点触控、滑动等。

React Native 的组件默认不处理触摸事件，要处理触摸事件，首先要向系统的“申请”成为触摸事件的响应器统一 (Responder)，当系统完成事件处理以后会释放响应器统一的角色。一个触摸事件处理周期，是从用户手指按下屏幕，到用户抬起手指结束。

3.5.2 手势系统响应

React Native 也提供了两个 API 来处理手势触控逻辑: GestureResponder 和 PanResponder。其中 GestureResponder 是底层的一个接口，而 PanResponder 是内置的手势识别库，提供了很多实用的手势识别接口。要了解 React Native 的手势响应系统就必须先了解 GestureResponder (手势响应)。

移动设备触摸操作背后的技术是相当复杂的，大多数移动设备都支持多点触控，这意味着在同一时刻系统需要处理多个有效的触摸点。当然多点触控有点复杂，所以系统提供了抽象的 Touchable 实现，通过实施正确的处理方法，视图可以成为接触响应器。前面讲过 TouchableHighlight 组件扮演响应器的角色，当然开发者也可以自己实现触摸响应器。如果一个视图想要变为响应器，需要实现以下几个方法：

- View.props.onStartShouldSetResponder
- View.props.onMoveShouldSetResponder
- View.props.onResponderGrant
- View.props.onResponderReject

为了更加形象地说明手势系统的工作流程，可以用如图 3-12 所示的流程图来说明。

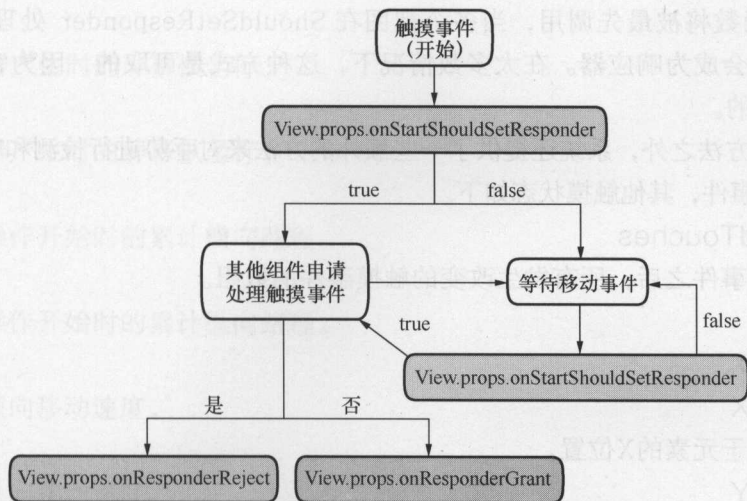


图3-12 响应器工作流程

一般情况下，组件需要与 Touchable 组件配合才能实现触摸操作，一个正常的触摸响应事件流程应该是这样的：是否接受响应→响应触摸事件→释放触摸事件。在 React Native 的手势响应系统中，一个完整的触摸事件分为 3 个生命周期状态：开始（Start）、移动（move）和释放（release），对应 Web 浏览器的 mouseDown、MouseMove 和 mouseUp 3 个生命周期状态。具体来讲，一个视图可以在开始或者移动阶段请求成为触摸事件的响应器，这些动作通过 onStartShouldSetResponder 和 onMoveShouldSetResponder 来指定。当视图返回为 true，说明视图申请响应触摸事件成功将进入相应的生命周期函数。

如果视图正在响应，那么会触发下列事件函数。

- View.props.onResponderMove:(evt)= > { }
用户正在移动手指。
- View.props.onResponderRelease:(evt)= > { }
在触摸结束被调用，即“touchUp”。
- View.props.onResponderTerminationRequest:(evt)= > true
其他元素想成为响应器，是否释放应答？返回 true 表示允许释放。
- View.props.onResponderTerminate:(evt)= > { }
响应器已经被回收。它可能被其他视图通过调用 onResponderTerminationRequest 之后被收回，也可能被系统强制收回（如 iOS 控制中心）。

有时候父组件想要成为响应器，就必须防止子视图成为应答器，所以在处理父组件时，onStartShouldSetResponderCapture 返回 true，视图就会尝试去得到应答器状态。当视图获得了应答器的状态后（可能失败，也可能成功）后，就会调用合适的回调函数，可能是 onResponderGrant，也可能是 onResponderReject，判断函数执行的方法叫做冒泡模式。

在冒泡模式下，最深的节点最先被调用，onStartShouldSetResponder 和 onMoveShould

SetResponder 函数将被最先调用，当多个视图在 ShouldSetResponder 处理程序返回 true 时，最深的组件会成为响应器。在大多数情况下，这种方式是可取的，因为它确保了所有控件和按钮是可用的。

除了上面的方法之外，系统还提供了一些额外的方法来对手势进行检测和响应。应答器是一个综合的响应事件，其他触摸状态如下。

- `changedTouches`
自从上个事件之后，所有发生改变的触摸事件的数组。
- `identifier`
触摸的 ID。
- `locationX`
触摸相对于元素的 X 位置。
- `locationY`
触摸相对于元素的 Y 位置。
- `pageX`
触摸相对于屏幕的 X 位置。
- `pageY`:
触摸相对于屏幕的 Y 位置。
- `target`:
接收触摸事件的元素的节点 ID
- `timestamp`
触摸的时间标识符，用于速度计算。
- `touches`
所有当前在屏幕上触摸的数组。

PanResponder

手势响应系统中，除了上面提到的 GestureResponder 之外，还需要对 PanResponder 有一定的了解。PanResponder 是 React Native 的一个类，它提供了处理原生相对高层的接口，用来处理更为复杂的触摸操作，例如多点触摸手势。对于每一个处理函数，它在原生事件之外提供了一个新的 `gestureState` 对象。`gestureState` 对象里面的字段可以帮助开发者处理更加复杂的触摸状态。PanResponder 提供的常见的触摸状态如下。

- `stateID`
触摸状态的 ID，在屏幕上有至少一个触摸点的情况下，这个 ID 会一直有效。
- `moveX`
最近一次移动时的屏幕横坐标。
- `moveY`
最近一次移动时的屏幕纵坐标。

- x0
当响应器产生时的屏幕横坐标。
- y0
当响应器产生时的屏幕纵坐标。
- dx
从触摸操作开始时的累计横向路程。
- dy
从触摸操作开始时的累计纵向路程。
- vx
当前的横向移动速度。
- vy
当前的纵向移动速度。
- numberActiveTouches
当前在屏幕上的有效触摸点的数量。

为了在组件中使用 PanResponder，我们需要创建一个 PanResponder 实例，然后将它添加到 render 的组件中。

先创建一个 PanResponder 实例。

```

this.myPanResponder = PanResponder.create({
  // 要求成为响应器统一：
  onStartShouldSetPanResponder: (evt, gestureState) => true,
  onStartShouldSetPanResponderCapture: (evt, gestureState) => true,
  onMoveShouldSetPanResponder: (evt, gestureState) => true,
  onMoveShouldSetPanResponderCapture: (evt, gestureState) => true,
  onPanResponderTerminationRequest: (evt, gestureState) => true,
  // 响应对应事件后的处理：
  onPanResponderGrant: (evt, gestureState) => {
    this.state.eventName=' 触摸开始 ';
    this.forceUpdate();
  },
  onPanResponderMove: (evt, gestureState) => {
    var _pos = 'x:' + gestureState.moveX + ',y:' + gestureState.moveY;
    this.setState( {eventName:' 移动 ',pos : _pos} );
  },
  onPanResponderRelease: (evt, gestureState) => {
    this.setState( {eventName:' 抬手' } );
  },
  onPanResponderTerminate: (evt, gestureState) => {
    this.setState( {eventName:' 另一个组件已经成为了新的响应器统一' } )
  },
});

```

然后，将 PanResponder 添加到 render 方法的组件中。

```
render() {
  return (
    <View style={styles.container} {...this.myPanResponder.panHandlers}>
      <Text>eventName:{this.state.eventName}|{this.state.pos}</Text>
    </View>
  );
}
```

之后，如果你的触摸起始于视图内，那么 PanResponder.create 的处理函数将会在相应的手势移动中被调用。

3.5.3 辅助功能

计算机辅助系统（Computer-aided system）是利用计算机辅助完成不同类任务的系统的总称。计算机辅助系统主要包含：计算机辅助设计（CAD）、计算机辅助制造（CAM）、计算机辅助工程（CAE）、计算机辅助测试（CAT）和计算机辅助教学（CAI）等。在手机等移动设备中，辅助系统也是不可缺少的一部分。

对 Android 系统而言，辅助功能涉及了许多不同的话题，其中之一是让丧失视力的人能够使用应用程序感知外部世界，谷歌提供了一个名叫 TalkBack 的内置屏幕读者服务机器人，借助该机器人，盲人可以通过触摸来使用移动设备和应用程序，TalkBack 可以使用文本语音转换器来阅读屏幕上的内容，并且可以发出警报来通知用户有关应用程序中的重要信息。

而对 iOS 系统而言，辅助功能也涵盖了许多话题，但对大多数人来说辅助功能就是 VoiceOver 的代名词，即 iOS 3.0 版本推出的一种语音辅助程序。它充当屏幕阅读器的角色，帮助有视觉障碍的人更方便地使用 iOS 设备。

辅助功能属性

如果视图是辅助功能元素，它把它的子元素分组成一个单一的可选组件，默认情况下，可触摸的所有元素都具有辅助性。

在 Android 系统中，react-native 视图中 accessible={true} 属性会被翻译成本地命令 focusable={true}。例如：

```
<View accessible={true}>
  <Text>text one</Text>
  <Text >text two</Text>
</View>
```

在上面的例子中，当父视图开启无障碍属性后，将不能获得 text one 和 text two 的辅助焦点。但是，我们可以在父元素上使用 accessible 属性来获得焦点。

accessibilityLabel(Android、iOS)

如果要将视图标记为具有辅助性，那么一个比较好的做法就是为这个视图设置一个

accessibilityLabel 标签，以便让使用 VoiceOver 的人知道他们选择了什么元素。当用户选择某个元素之后，VoiceOver 将会阅读响应的字符串文本。使用 accessibilityLabel 时，将视图中的 accessibilityLabel 属性设置为一个自定义的字符串即可。例如：

```
<TouchableOpacity accessible={true} accessibilityLabel={'Tap me!'}
  onPress={this._onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Press me!</Text>
  </View>
</TouchableOpacity>
```

在上面的例子中，TouchableOpacity 元素中的 accessibilityLabel 会被默认设置为“Press me!”。该标签通过使用空格符来串联所有文本节点子元素。

accessibilityTraits (iOS)

辅助功能告诉用户在使用 VoiceOver 的时候选择了什么元素，以及相关的内容信息，accessibilityTraits 常用的辅助字符串如下。

- none
当元素没有特征的时候使用。
- button
当元素需要被当作按钮的时候使用。
- link
当元素需要被当做链接的时候使用。
- header
当元素作为内容部分的标题（如导航栏中的标题）的时候使用。
- search
当文本字段元素被视为一个搜索字段的时候使用。
- image
当元素被作为图像的时候使用，可以和按钮或链接等连用。
- selected
当该元素被选中时使用。例如，表中被选中的行。
- plays
当元素被激活并且播放自己的声音的时候使用。
- key
当元素充当键盘按键的时候使用。
- text
当元素被视为不能更改的静态文本的时候使用。
- summary
当在应用程序首次启动的时候，该元素可以提供应用程序的实时状况的时候使用。例

如，天气预报应用程序首次启动的时候，带有当天天气信息的元素将被该特征所标记。

- **disabled**
当控件未启动并且对用户的输入无响应的时候使用。
- **frequentUpdates**
当元素经常更新其标签或者值的时候，允许辅助功能客户端隔一段时间再去检查变化，避免频繁的更新操作。例如，秒表就是典型的例子。
- **startsMedia**
当激活元素并开始一段媒体会话（例如播放电影，录制音频）时，不会被辅助技术的输出所打断（如VoiceOver）。
- **adjustable**
当元素可以被“调整”的时候使用。
- **allowsDirectInteraction**
当元素允许VoiceOver用户直接进行触摸互动的时候使用。
- **pageTurn**
当它完成阅读的元素的内容需要通知VoiceOver滚动到下一个页面。

onAccessibilityTap (iOS)

此属性用来分配一个自定义的函数，当用户双击某个选中的元素时调用该函数。

onMagicTap (iOS)

当有人使用两个手指双击的时候，该属性就会被分配给一个自定义函数，同时，这个函数会被调用。在 iPhone 手机应用程序中，使用 magic tap 敲击可以接听或者结束一个电话。如果所选的元素不具有 onMagicTap 功能，该系统将遍历视图层次结构，直到它找到一个拥有此功能的视图为止。

accessibilityComponentType (Android)

在某些情况下，用户被提醒选定的组件类型（比如按钮），如果我们使用的是原生组件，这一行为会自动进行。由于我们使用 JavaScript 来开发应用程序，所以真正开发的时候就需要为 Android 环境下的 TalkBack 提供更多的使用环境。例如，让按钮支持 button、radiobutton_checked 和 radiobutton_unchecked 等属性，那么，就需要对 UI 组件的 accessibilityComponentType 属性做相关的声明。示例代码如下：

```
<TouchableWithoutFeedback accessibilityComponentType="button"
  onPress={this._onPress}>
  <View style={styles.button}>
    <Text style={styles.buttonText}>Press me!</Text>
  </View>
</TouchableWithoutFeedback>
```

accessibilityLiveRegion (Android)

当组件状态发生更改时，我们希望 TalkBack 能发出通知提醒用户，可以通过设置

AccessibilityLiveRegion 属性来实现这一诉求。它提供了诸如 none、polite 和 assertive 等属性。

- none
辅助功能服务不应该对此视图通知改变的地方。
- polite
辅助功能服务应该对此视图通知改变的地方。
- assertive
辅助功能服务应该中断正在进行的会话，并且以立即宣布该视图的改变。

```
<TouchableWithoutFeedback onPress={this._addOne}>
  <View style={styles.embedded}>
    <Text>Click me</Text>
  </View>
</TouchableWithoutFeedback>
<Text accessibilityLiveRegion="polite">
  Clicked {this.state.count} times
</Text>
```

在上面的例子中，通过监听 _addOne 的状态来监听用户的行为，当最终用户单击的时候，因为 TalkBack 设置了 accessibilityLiveRegion="polite" 属性，所以当它读取了文本视图中的文本后，就会发出视图改变的通知。

importantForAccessibility (Android)

有时候，面对两个重叠并且拥有相同父元素的组件，默认的辅助功能焦点行为往往是不可预知的，这时候就要通过 ImportantForAccessibility 属性来控制解决问题。importantForAccessibility 提供的常用功能属性有 auto、yes、no 以及 no-hide-descendants。示例代码如下：

```
<View style={styles.container}>
  <View style={{backgroundColor: 'green'}} importantForAccessibility="yes">
    ...
  </View>
  <View style={{backgroundColor: 'yellow'}}
    importantForAccessibility="no-hide-descendant">
    ...
  </View>
</View>
```

3.6 小结

本章主要对 React Native 开发中涉及的一些基础知识和概念进行了一个简单的介绍。通过这些知识的介绍，以及和 Web 开发环境的比较，我们对 React Native 页面开发中涉及的布局、样式以及基础页面的绘制和事件等有了一个全面的了解。下一章将对 React Native 开发中使用频率比较高的常用组件做详细介绍。

常用组件介绍

在上一章中，主要对 React Native 的基础语法和样式做了一些简单的介绍。在本章中将详细介绍 React Native 开发中比较基础的移动组件：`<View>`、`<Text>`、`<Button>`、`<Image>`、`<ListView>`、`<Navigator>` 等，然后探讨下 React Native 开发中与触摸事件相关的系列组件。

4.1 HTML 元素与原生组件

在 Web 程序开发中，我们使用各种基础的 HTML 元素来开发页面，如 `<html>`、`<div>`、`<head>` 等，各种格式标签，如 `
`、`<dt>`、`` 等。而在 React Native 开发中，我们不能使用 HTML 标签，而是使用系统提供的组件来开发界面，如 `<View>`、`<Text>`、`<Button>` 等，它们之间的对比如表 4-1 所示。

表 4-1

HTML 元素与 React Native 组件对比

HTML	React Native
div	View
img	Image
span, p	Text
li, ol	ListView

虽然，这些元素有很多相似之处，但是它们不能相互替换使用。那么，React Native 基础组件有没有办法渲染成 HTML 元素呢？暂时是不能的，HTML 是基于元素标签来开发页面，而 React Native 是基于组件来开发页面，React Native 组件目前只能在 iOS 和 Android 平台进行复用。然而，除了基础组件之外，JavaScript 代码是可以复用的。下面，通过对比，让我们来了解一下 HTML 元素与 React Native 组件的区别。

4.1.1 文本组件

文本组件是 React Native 开发中最基本的组件之一，几乎任何应用都需要用到它。然而，React Native 的文本组件和 HTML 的文本标签却大不相同，它们使用上也有所差异。

在 HTML 中处理文本时，开发人员可以使用像 ``、`<p>` 这样的子标签给文本标签添加样式。例如：

```
<html>
...
  <strong>This text is strong</strong>
</html>
```

在 React Native 中，只有 `<Text>` 组件能作为纯文本的子节点，换句话说，必须使用 `<Text>` 组件将内容包起来。文本不使用 HTML 子标签，而是使用文本相关属性。例如：

```
<Text>
The quick <Text style={{fontStyle:"italic"}}>Text Style</Text>
</Text>
```

但是，上面这种代码看起来非常不清楚，显得有点冗长。这时候，创建样式组件来处理文本就显得非常必要了。例如：

```
var styles=StyleSheet.create({
  bold:{fontWeight:"bold"},
  italic:{fontStyle:"italic"}
});
```

```
var Strong=React.createClass({
  render:function(){
    return(
      <Text style={styles.bold}>
        {this.props.children}
      </Text>);
  }
});
```

一旦申明样式组件，就可以自由地嵌套样式了。可能你已经发现了，React Native 更提出的是样式组件的复用，而不仅仅是样式的复用。

4.1.2 图片组件

如果说在一个应用中，文本是最基本的元素之一，那么图像在移动端和 Web 端也是最重要的元素之一。在 Web 程序开发中，有时使用 `` 标签来添加图片，有时通过 CSS 方式来导入图片，如 `background-image` 属性。而在 React Native 中，我们使用 `<Image>` 组件来导入图片。

加载本地图片

`<Image>` 组件的使用方法也比较简单，只需要设置 `source` 属性即可。例如：

```
<Image style={styles.img} source={require('./image/xxx.png')} />
```

在 React Native 开发中，不允许使用字符串变量来指定预加载的图片的地址，因为 React Native 是在编译时处理所有的 `require` 声明，而不是在运行时动态地处理。所以下面的写法是错误的：

```
var imageAddress = './image/xxx.png';
class AwesomeProject extends Component {
  render() {
    return (
      <View style={styles.container}>
        // 错误的引用
        <Image style={styles.imageStyle}
          source={require(imageAddress)} />
      </View>
    );
  }
}
```

加载网络图片

加载网络图片和加载本地图片类似，唯一的区别在于加载网络图片的 `source` 是网络地址，所以要在图片的网络地址前加上 `uri` 标签。例如：

```
var imageAddress = 'http://ohe65w0xx.bkt.clouddn.com/react.png';
class AwesomeProject extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Image style={styles.imageStyle}
          source={{uri:imageAddress}} />
      </View>
    );
  }
}
```

resizeMode

在 React Native 中, Image 组件必须在样式中声明图片的宽和高属性, 如果不设置, 则不会显示。一般我们将 Image 定义的宽和高乘以当前运行环境的像素密度, 来作为 Image 实际的宽和高。当 Image 的实际宽、高与图片的实际宽、高不符时, 通常使用 resizeMode 来进行调整。resizeMode 提供的属性值主要有 3 种: contain、cover 和 stretch。它们之间的主要区别如下。

- cover: cover 模式只求在显示比例不失真的情况下填充整个显示区域。可以对图片进行放大或者缩小, 超出显示区域的部分不显示。
- contain: contain 模式是要求显示整张图片, 可以对它进行等比缩小。如果图片宽高都小于控件宽高, 则不会对图片进行放大。
- stretch: stretch 模式不考虑保持图片原来的宽高比, 会填充整个 Image 定义的显示区域, 这种模式下的图片可能会畸形和失真。

为了更加直观地观察 3 种模式的区别, 我们看一个例子: 引入同一张图片, 使用不同的 resizeMode, 图片显示会有所区别。

```
export default class Demo extends Component {
  componentWillMount() {
    this.image=reqUIre('./image/react.png');
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          React Native 图片 resizeMode 属性
        </Text>
        <Image style={[styles.imageStyle,{resizeMode:'cover'}]}
          source={this.image}/>
        <Text style={styles.textStyle}>cover</Text>
        <Image style={[styles.imageStyle,{resizeMode:'contain'}]}
          source={this.image}/>
        <Text style={styles.textStyle}>contain</Text>
        <Image style={[styles.imageStyle,{resizeMode:'stretch'}]}
          source={this.image}/>
        <Text style={styles.textStyle}>stretch</Text>
      </View>
    );
  }
}
```

上面代码的运行效果如图 4-1 所示。

其他样式

除了上面讲到的一些常用的样式之外, Image 组件支持绝大多数的 View 样式属性, 下面是一些常用的属性列出来。

- backfaceVisibility ['visible', 'hidden']
隐藏或者显示。
- backgroundColor color
背景色。
- borderBottomLeftRadius、borderBottomRightRadius、borderTopLeftRadius、borderTopRightRadius
角边圆角大小。
- borderColor color
边框颜色。
- borderRadius
边框圆角。
- borderWidth number
边框宽度。
- opacity
设置透明度。
- overflow ['visible', 'hidden']
设置图片尺寸超过容器是否显示或者隐藏。
- resizeMode
图片调整模式。
- tintColor color
颜色设置。
- overlayColor(Android)
当图片圆角显示时，剩余空间设置的颜色，Android独有。

Image 组件是我们开发中使用比较多的组件，你可以根据实际项目需要做相应的选择。

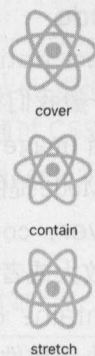


图4-1 不同resizeMode属性之间的区别

4.1.3 TextInput组件

输入框组件在应用开发中是必不可少的，它允许用户在应用中通过键盘输入文本信息或富文本信息。在 React Native 中，TextInput 组件提供了丰富的功能，例如自动拼写修复、自动大小写切换、占位默认字符设置以及多种不同类型的键盘切换等。

TextInput简单使用

例如，使用 TextInput 实现一个简单的登录页面。不过需要注意的是，要让 Text 组件的文本居中，需要在 Text 之外添加一个 View 层。

```
<View style={styles.styleSubmit}>
  <Text style={styles.submit}>
    登录
  </Text>
</View>
```

```

</View>
// 省略...
const styles = StyleSheet.create({
  styleSubmit: {
    // 省略...
    justifyContent: 'center',
    alignItems: 'center',
  },
});

```

运行效果如图 4-2 所示。

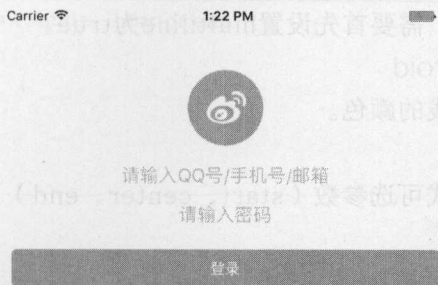


图4-2 登录界面

TextInput属性和方法

TextInput 提供了丰富的属性及方法，如下所示。

- autoCAPitalize

控制TextInput是否自动将特定字符切换为大写。可选择参数none、sentences、words、characters。区别如下：

none: 不自动切换任何字符成大写。

sentences: 默认每个句子的首字母变成大写。

words: 每个单词的首字母变成大写。

characters: 每个字母全部变成大写。

- placeholder

占位符，在输入前显示的文本内容。

- autoCorrect

拼写自动修正功能默认为开启（true）。

- autoFocus

文本焦点获取，默认为关闭（false）。

- maxLength

设置文本的最大长度。

- `editable`
设置文本是否可输入。
- `keyboardType`
设置键盘显示类型。选择参数：`default`、`email-address`、`numeric`、`phone-pad`、`ascii-capable`、`numbers-and-punctuation`、`url`、`number-pad`、`name-phone-pad`、`decimal-pad`、`twitter`、`Web-search`。
- `multiline`
设置可以输入多行文字，默认为`false`。
- `numberOfLines`
设置文本输入框行数，需要首先设置`multiline`为`true`。
- `underlineColorAndroid`
设置文本输入框下划线的颜色。
- `textAlign`
设置文本横向布局方式可选参数（`start`、`center`、`end`）。
- `onBlur()`
文本框失去焦点监听回调方法。
- `onChange()`
文本框内容发生改变回调方法。
- `onChangeText()`
文本框内容发生改变回调方法，和`onChange()`不同的是，改变后的文件内容会作为参数传递。
- `onEndEditing()`
当文本输入结束，调用此回调方法。
- `onFocus()`
文本框获取到焦点回调方法。
- `onLayout()`
文本组件布局发生变化的时候调用，调用方法参数为 `{x,y,width,height}`。
- `onSubmitEditing()`
编辑提交的时候回调方法。

TextInput示例

在很多应用类 APP 中，为了方便用户精准地查找产品，都会提供搜索功能，当用户输入搜索关键字后，前端 APP 应用程序通过接口向后端发送相关请求，后台接到请求后，通过接口返回相关的搜索结果内容给应用程序使用。不过在本例中，为了讲解方便，直接使用本地内置的数据。

如图 4-3 所示，要实现上面的效果，可以通过对 `TextInput` 的 `onChangeText()` 输入文本的监听来实现，同时在该方法里面给出搜索结果，最后以列表的方式展示出来。

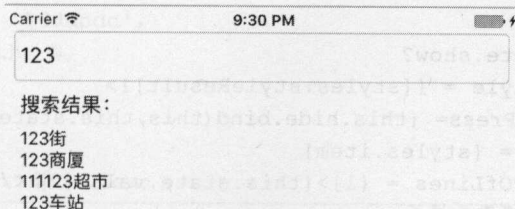


图4-3 TextInput监听文本输入

示例代码如下：

```
class TextInputView extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
      text: ''
    };
  }
```

```
  hide(val) {
    this.setState({
      show: false,
      value: val
    });
  }
```

```
  getValue(text) {
    var value = text;
    this.setState({
      show: true,
      value: value
    });
  }
```

```
  render() {
    return (
```

```
      <View style={styles.container}>
        <TextInput style = {styles.styleInput}
          returnKeyType = "search"
          placeholder= " 请输入搜索关键字 "
          onEndEditing = {this.hide.bind(this,this.state.value)}
          value = {this.state.value}
          onChangeText = {this.getValue.bind(this)}>/>
```

```
      <Text style={styles.styleText}> 搜索结果:</Text>
```

```

    {this.state.show?
      <View style = {[styles.styleResult]}>
        <Text onPress= {this.hide.bind(this,this.state.value + ' 街')}
          style = {styles.item}
          numberOfLines = {1}>{this.state.value} 街 </Text>
        // 省略 n 多默认提示
        <Text onPress = {this.hide.bind(this,this.state.value + ' 车站')}
          style = {styles.item}
          numberOfLines = {1}>{this.state.value} 车站 </Text>
        </View>:null}
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#ffffff',
    marginTop:20
  },
  styleInput: {
    height: 40,
    borderWidth: 1,
    marginLeft: 10,
    marginRight:10,
    paddingLeft: 5,
    borderColor: '#cccccc',
    borderRadius: 4,
  },
  styleResult: {
    marginTop: 10,
    marginLeft: 15,
  },
  styleText: {
    fontSize: 16,
    marginTop:10,
    marginLeft:15
  },
  styleItem: {
    fontSize: 18,
    padding: 5,
    paddingTop: 10,
    paddingBottom: 10,
    borderWidth: 1,
    marginLeft:15,
  }
});

```

```
borderColor: '#dddddd',
borderTopWidth: 0,
}
});
```

```
export default TextInputView;
```

4.1.4 ScrollView组件

当需要展示的内容较多、超出一个屏幕时，用户可以通过滚动窗口来查看屏幕以外的内容。在 React Native 中，可以通过将子组件包裹在 ScrollView 内部来达到滚动的效果，不过需要注意的是，ScrollView 必须有一个确定的高度才能正常工作，因为它实际上就是将一系列不确定高度的子组件装进一个确定高度的容器内。由于 ScrollView 还集成了触摸锁定的“响应器统一”系统，所以，ScrollView 内部的其他响应器尚无法阻止 ScrollView 本身成为响应器。

在原生应用开发中，经常会有人问到 ScrollView 和 ListView/FlatList 之间的联系和区别，因为 ListView 在某些条件下也提供滚动的效果。简单来说，ScrollView 会简单粗暴地把所有子元素一次性全部渲染出来，而 ListView 会惰性渲染子元素，只有当它们将要出现在屏幕上时才开始渲染。从 0.43 版本开始，React Native 为开发者提供了改进型的 ListView 组件 FlatList，性能更优，体验更佳。

ScrollView属性

ScrollView 常用的属性如下所示。

- `contentContainerStyle` `StyleSheetPropType(ViewStylePropTypes)`
使用一个样式作为内层的内容容器的样式，所有的子视图都会包裹在内容容器内。
- `horizontal`
系统默认在垂直方向上滚动。如果此属性为 true，所有的子视图会在水平方向上滚动。
- `keyboardDismissMode`
枚举类型，用户拖拽滚动视图的时候，是否要隐藏软键盘。提供 none、interactive、on-drag 选项。区别如下。
none：默认值，拖拽时不隐藏软键盘。
interactive：拖拽开始的时候隐藏软键盘。
on-drag：软键盘伴随拖拽操作同步消失，并且如果往上滑动会恢复键盘。
- `keyboardShouldPersistTaps`
当此属性为 false 的时候，在软键盘激活之后，点击焦点文本输入框以外的地方，键盘就会隐藏。如果为 true，滚动视图不会响应点击操作，并且键盘不会自动消失。
- `refreshControl`
指定 RefreshControl 组件，用于为 ScrollView 提供下拉刷新功能。
- `removeClippedSubviews`
当此属性为 true 时，屏幕之外的视图会被移除（即 ScrollView 不会作用于隐藏的部分）。

- `showsHorizontalScrollIndicator/ showsVerticalScrollIndicator`
是否显示水平/垂直滚动条。
- `pagingEnabled`
当值为`true`时，滚动条会停在滚动视图的尺寸的整数倍位置。默认值为`false`。
- `scrollEnabled`
当值为`false`的时候，内容不能滚动，默认值为`true`。
- `stickyHeaderIndices`
决定子视图在滚动之后固定在屏幕顶端。例如，`stickyHeaderIndices=[0]`会让第一个成员固定在滚动视图顶端。

ScrollView方法

ScrollView 常用的方法如下所示。

- `onContentSizeChange()`
当ScrollView内部可滚动内容的视图发生变化时调用。
- `onScroll()`
在滚动的过程中调用。调用的频率可以用`scrollEventThrottle`属性来控制。
- `scrollTo({x, y, animated})`
滚动到指定的`x`、`y`偏移处，第三个参数为是否启用平滑滚动动画。例如，`scrollTo({x: 0, y: 0, animated: true})`。
- `scrollToEnd({x, y, animated})`
滚动到视图底部（水平方向的视图则滚动到最右边），第三个参数为是否启用平滑滚动动画。

ScrollView示例

在很多应用型 APP 中，广告轮播是必不可少的，广告位会推送一些最及时的活动。在 React Native 中，要想实现这种效果，可以使用 ScrollView 来实现。当然，为了方便，我们也可以使用第三方库，例如 `react-native-swiper`。首先，看一下使用 ScrollView 实现广告轮播的效果，如图 4-4 所示。



图4-4 使用ScrollView组件实现广告轮播

图中整个广告轮播界面主要由底图和指示点构成。要实现这样的效果，首先要有数据源，然后利用 ScrollView 的相关属性和方法绘制界面和处理滚动逻辑。

首先，准备数据源图片数组。

```
var imageData = ['https://img3.doubanio.com/view/movie_poster_cover/mpst/public/p2263582212.jpg', 'https://img3.doubanio.com/view/movie_poster_cover/mpst/public/p2265761240.jpg', 'https://img3.doubanio.com/view/movie_poster_cover/mpst/public/p2266110047.jpg'];
```

然后，我们需要设置 ScrollView 的相关属性，并给 ScrollView 绑定滚动监听事件。

```
horizontal={true}
showsHorizontalScrollIndicator={false}
pagingEnabled={true}
onMomentumScrollEnd={(e) => {
  this.onAnimationEnd(e)
}}
```

接着，根据数据源渲染界面和指示点。

```
// 渲染界面
renderImages() {
  let allImage = [];
  for (let i = 0; i < imageData.length; i++) {
    let item = imageData[i];
    allImage.push(
      <Image key={i} source={{uri: item}} style={styles.imageStyle}/>
    );
  }
  return allImage;
}

// 渲染指示点
renderPagingIndicator() {
  let indicatorArr = [];
  let style;
  for (let i = 0; i < imageData.length; i++) {
    style = (i == this.state.currentPage) ? {color: 'orange'} : {color: 'white'};
    indicatorArr.push(
      <Text key={i} style={{fontSize: 30, style}}>
        .
      </Text>
    );
  }
  return indicatorArr;
}
```

到此，完整的广告轮播功能就完成了。完整代码如下：

```

var Dimensions = require('Dimensions');
var screenWidth = Dimensions.get('Window').width;

var imageData = ['https://img3.doubanio.com/view/mpst/public/p2263582212.jpg', 'https://img3.doubanio.com/view /mpst/public/p2265761240.jpg', 'https://img3.doubanio.com/vie /mpst/public/p2266110047.jpg'];

class HorizontalScrollView extends Component {

  constructor(props) {
    super(props);
    this.state = {currentPage: 0};
  }

  renderImages() {
    let allImage = [];
    for (let i = 0; i < imageData.length; i++) {
      let item = imageData[i];
      allImage.push(
        <Image key={i} source={{uri: item}} style={styles.imageStyle}/>
      );
    }
    return allImage;
  }

  onAnimationEnd(e) {
    let offsetX = e.nativeEvent.contentOffset.x;
    let pageIndex = Math.floor(offsetX / screenWidth);
    this.setState({currentPage: pageIndex});
  }

  renderPagingIndicator() {
    let indicatorArr = [];
    let style;
    for (let i = 0; i < imageData.length; i++) {
      style = (i == this.state.currentPage) ? {color: 'orange'} : {color: 'white'};
      indicatorArr.push(
        <Text key={i} style={[{fontSize: 30}, style]}>
          .
        </Text>
      );
    }
    return indicatorArr;
  }

  render() {

```

```

    return (
      <View style={styles.container}>
        <ScrollView
          ref='scrollView'
          horizontal={true}
          showsHorizontalScrollIndicator={false}
          pagingEnabled={true}
          onMomentumScrollEnd={(e) => {
            this.onAnimationEnd(e)
          }}
        >
          {this.renderImages()}
        </ScrollView>
        <View style={styles.pagingIndicatorStyle}>
          {this.renderPagingIndicator()}
        </View>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    marginTop: 20,
    backgroundColor: '#ffffff'
  },
  scrollViewStyle: {
    backgroundColor: 'yellow',
  },
  imageStyle: {
    width: screenWidth,
    height: 200,
  },
  pagingIndicatorStyle: {
    height: 25,
    width: screenWidth,
    backgroundColor: 'rgba(0,0,0,0)',
    position: 'absolute',
    bottom: 0,
    flexDirection: 'row',
    alignItems: 'center',
    justifyContent: 'center',
  }
});

export default HorizontalScrollView;

```


4.2 结构化组件

在传统的 Android、iOS 开发中，页面开发必然会用到很多的组件。组件是组成页面最基本的元素，而在这些组件中，有一些组件起到导航的作用，控制着应用的总体流向。这些组件包括 <View>、<Navigator>、<ListView> 以及 <TabView> 等。

4.2.1 View 组件

作为创建 UI 时最基础的组件，View 是一个支持 Flexbox 布局、样式、一些触摸处理和一些无障碍功能的容器，并且它可以放到其他的视图组件里，也可以有任意多个任意类型的子视图。无论是 iOS、Android 还是 Web 平台，View 都会直接对应平台的原生视图，其作用等同于 iOS 平台下的 UIView、Web 平台下的 <div> 和 Android 平台下的 View。

View 简单使用

例如，创建一个视图，将 3 种不同颜色的框排成一行，第一个占 50%，其余两个各占 25%。

```
<View style={{flexDirection: 'row', height: 100, padding: 20}}>
  <View style={{backgroundColor: 'blue', flex: 0.5}} />
  <View style={{backgroundColor: 'red', flex: 0.25}} />
  <View style={{backgroundColor: 'green', flex: 0.25}} />
</View>
```

运行效果如图 4-5 所示。

View 组件常用属性

View 常用的属性和方法如下所示。

- accessibilityLabel

隐藏或者显示设置当用户与此元素交互时，“读屏器”（对视力障碍人士的辅助功能）阅读的文字。

- accessible

当此属性为 true 时，表示此视图是一个启用了无障碍功能的元素。

- onAccessibilityTap

当 accessible 为 true 时，如果用户对一个已选中的无障碍元素做了一个双击手势，系统会调用此函数（注：此事件是针对残障人士，并非是一个普通的点击事件）。

- onLayout

当组件挂载或者布局变化的时候调用。当事件被调用时，新的布局可能还没有在屏幕上呈现。

- onMagicTap

当 accessible 为 true 时，如果用户做了一个双指轻触（magic tap）手势，系统会调用此函数。

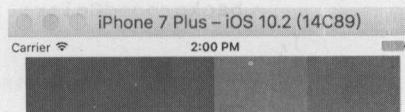


图4-5 View 示例

- `onResponderGrant`

视图正在响应触摸事件时调用此函数。

- `pointerEvents`

用于控制当前视图是否可以作为触控事件的目标，它是一个枚举类型的属性。枚举值主要有：`box-none`、`none`、`box-only`、`auto`。区别如下。

- `auto`视图可以作为触控事件的目标。

- `none`视图不能作为触控事件的目标。

- `box-none`视图自身不能作为触控事件的目标，但其子视图可以。

- `accessibilityComponentType(Android)`

UI组件与原生组件是否一致处理，本属性仅针对Android。

- `accessibilityLiveRegion (Android)`

当view发生更新的时候，是否要通知用户，仅针对Android API ≥ 19 的设备。

- `renderToHardwareTextureAndroid (Android)`

这个属性用来设置是否需要使用GPU进行渲染。本属性仅针对Android。

- `shouldRasterizeIOS(iOS)`

定义该UI组件与原生组件是否一致处理，仅针对iOS。

View组件使用

View 的设计初衷是和 `StyleSheet` 搭配使用，这样可以使代码更清晰并且获得更高的性能。例如，使用 `StyleSheet` 来实现图 4-5 的例子。

```
render() {
  return (
    <View style={styles.containerStyle}>
      <View style={styles.childBlueStyle} />
      <View style={styles.childRedStyle} />
      <View style={styles.childGreenStyle} />
    </View>
  );
}

...

const styles = StyleSheet.create({
  containerStyle: {
    flexDirection: 'row',
    height: 100,
    padding: 20
  },
  childBlueStyle: {
    backgroundColor: 'blue',
    flex: 0.5
  },
  childRedStyle: {
```

```

        backgroundColor: 'red',
        flex: 0.25
      },
      childGreenStyle: {
        backgroundColor: 'green',
        flex: 0.25
      },
    },
  });

```

View 组件是 React Native 开发中最基本的组件，也是我们平时开发中使用频率比较高的组件之一。

4.2.2 ListView组件

ListView 作为核心组件之一，主要用于高效地显示一个可以垂直滚动的变化的数据列表。经过自定义组装，我们还可以用它实现九宫格等页面效果。

在 React Native 中，使用 ListView 组件至少需要两个属性：DataSource 和 renderRow。DataSource 是需要渲染界面的数据源，renderRow 是根据数据源的元素返回的可渲染的组件，即 ListView 的一行。

在 React Native 中，最基本的使用方式就是创建一个 ListView.DataSource 数据源，然后给它传递一个普通的数据数组，再使用数据源来实例化一个 ListView 组件，并且定义它的 renderRow 回调函数，这个函数返回 ListView 的一行作为一个可渲染的组件。例如，下面是 React Native 官网提供的 ListView 的使用示例：

```

constructor(props) {
  super(props);
  var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
  this.state = {
    dataSource: ds.cloneWithRows(['第一行', '第二行', '第三行']),
  };
}
// 省略...
render() {
  return (
    <ListView
      dataSource={this.state.dataSource}
      renderRow={(rowData) => <Text>{rowData}</Text>}
    />
  );
}

```

运行效果如图 4-6 所示。

ListView常用属性

在讲 ListView 的高级用法之前，首先来了解下 ListView 组件的常用属性。

iPhone 7 Plus – iOS 10.2 (14C89)

第一行
第二行
第三行

图4-6 ListView示例效果

- **dataSource**
列表数据源，ListView.DataSource实例。
- **initialListSize**
指定在组件渲染多少行数据，常用于首屏推荐热门商品。
- **onChangeVisibleRows**
本属性用来回调可见的行的集合变化的情况，有两个参数：visibleRows和changedRows。visibleRows 以 { sectionID: { rowID: true }}的格式包含了所有可见行，而changedRows 以{ sectionID: { rowID: true | false }}的格式包含了所有刚刚改变了可见性的行。如果值为true，表示一个行变得可见，值为false，表示行刚刚离开可视区域而变得不可见。
- **onEndReached**
当所有数据都被渲染，并且列表被滚动到onEndReachedThreshold个的底部的时候被调用，原生的滚动事件会被作为参数进行传递。
- **removeClippedSubviews**
用于提升复杂列表的滚动性能，使用时需要给行容器添加“隐藏”样式，此属性默认开启。
- **renderRow**
(rowData、sectionID、rowID、highlightRow) => renderable从数据源中接受一条数据，根据它在数据源数组所在的位置，返回一个可渲染的组件来为这行数据进行渲染。可以通过highlightRow函数来通知ListView某一行高亮的效果。
- **renderSectionHeader**
调用此函数渲染一个粘性的标题，使用 stickySectionHeadersEnabled来决定是否启用其粘性特性。
- **renderSeparator**
用于渲染列表的分割线。
- **sticRHeaderindices**
一个子视图下标的数组，用于决定在滚动之后固定在屏幕顶端的元素，例如，stickyHeaderIndices=[0]会让第一个元素固定在滚动视图顶端。
- **stickySectionHeadersEnabled**
用于设置小节标题（section header）是否具有粘性。需要注意的是，此设置在水平模式

horizontal={true}下无效。此属性在iOS平台默认开启，Android平台默认关闭。

- scrollTo

调用此函数可以滚动到指定的x, y偏移处，可以指定是否加上过渡动画。

- scrollToEnd

调用此函数用于在垂直方向上滚动到视图底部，水平方向则滚动到最右边。加上动画参数的{animated: true}则启用平滑滚动动画，调用 scrollToEnd的{animated: false}) 则立即跳转到视图底部，animated动画选项默认启用。

Listview案例：商品列表

如图 4-7 展示了 ListView 商品列表，使用 ListView 控件时，需要注意 rowHasChanged 函数，它用来控制 ListView 是否需要重绘一行数据，当数据发生改变时，控件会通知系统重绘界面。

```
var ds = new ListView.DataSource({rowHasChanged: (r1, r2) => r1 !== r2});
```

在实际应用过程中，关于 ListView 的使用往往会涉及异步网络请求、复杂多变的列表子视图，以及处理控件嵌套过程中的兼容问题等。



图4-7 ListView商品列表展示

商品列表标题

商品标题就是一个 Text 控件，为了方便以后代码的维护，我们将常用的 Text 做了简单的样式进行归类，然后在使用地方导入控件样式即可。示例代码如下：

```
export function Heading1({style, ...props}: Object): ReactElement {
  return <Text style={[styles.h1, style]} {...props} />
```

```

}
// 省略...
const styles = StyleSheet.create({
  h1: {
    fontSize: 15,
    fontWeight: 'bold',
    color: '#222222',
  }
});

```

商品列表

列表由 ListView 渲染每一个子视图而形成，在每一个子视图中主要包含 3 个方面的信息：商品图片、标题、产品编号。所以对应的数据结构如下：

```

{id": "3419", // 商品 id
"pic": "https://facebook.github.io/react/img/logo_og.png", // 商品图片
"title": "米米的美淘杂货铺" // 商品标题

```

根据每个视图的字段，我们可以很容易地将子视图的界面绘制出来。

```

renderCell(cellData) {
  return (
    <TouchableOpacity onPress={()=>this.pressRow(cellData)}>
      <View>
        <View style={styles.cellContainer}>
          <Image
            style={styles.thumbnail}
            source={{uri:cellData.pic}}
          />
          <View style={styles.itemCellView}>
            <Text numberOfLines={1} style={styles.title}>{cellData.title}</Text>
            <Text numberOfLines={1} style={styles.id}>产品编号: {cellData.id}</Text>
          </View>
        </View>
        <Text style={styles.itemDivice}/>
      </View>
    </TouchableOpacity>
  );
}

```

接下来需要给 ListView 设置数据源。为了方便理解，数据源直接采用本地读取的模拟数据即项目 data 目录下的 json 模拟数据。在 React Native 的生命周期函数中，有一个 componentDidMount 函数，这个函数用来通知组件已经加载完成，我们可以在这个方法里面模拟数据请求操作。

```
let data=require('./data/list.json');
    this.setState({
        dataSource:this.state.dataSource.cloneWithRows(data.list),
    });
```

读取的本地 json 文件内容如下:

```
{
  "list": [
    {
      "id": "3419",
      "pic": "https://facebook.github.io/react/img/logo_og.png",
      "title": " 米米的美淘杂货铺 "
    },
    // 省略...
    {
      "id": "3426",
      "pic": "https://facebook.github.io/react/img/logo_og.png",
      "title": " 纽约时尚潮鞋总代理 "
    }
  ]
}
```

完整示例代码

关于网络 fetch 异步获取数据,将在后面的章节详细介绍。在本例中,数据源为本地的模拟数据。上面例子的完整代码如下:

```
// 省略 import 导包
```

```
export default class Demo extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
      dataSource:new ListView.DataSource({
        rowHasChanged:(r1,r2) => r1!=r2,
      }),
    }
  }
```

```
// 获取本地 json 数据
```

```
componentWillMount() {
  let data=reqUIre('./data/list.json');
  this.setState({
    dataSource:this.state.dataSource.cloneWithRows(data.list),
  });
}
```

```

render() {
  return (
    <View style={styles.container}>
      <StatusBarIos/>
      <View style={styles.recommendHeader}>
        <Heading2> 猜你喜欢 </Heading2>
      </View>
      <View style={styles.container}>
        <ListView style={styles.listView}
          dataSource={this.state.dataSource}
          renderRow={this.renderCell.bind(this)}
        />
      </View>
    </View>
  );
}

renderCell(cellData) {
  return (
    <TouchableOpacity onPress={()=>this.pressRow(cellData)}>
      <View>
        <View style={styles.cellContainer}>
          <Image
            style={styles.thumbnail}
            // source={{uri:'https://facebook.github.io/react/img/logo_
og.png'}}
            source={{uri:cellData.pic}}
          />
          <View style={styles.itemCellView}>
            <Text numberOfLines={1} style={styles.title}>{cellData.title}</
Text>
            <Text numberOfLines={1} style={styles.id}> 产品编号:
{cellData.id}</Text>
          </View>
        </View>
        <Text style={styles.itemDivice}/>
      </View>
    </TouchableOpacity>
  );
}

// 点击弹框
pressRow(data) {
  alert(" 点击了: "+data.id);
}
}

```



```

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#FFFFFF',
  },
  cellContainer: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    margin: 10
  },
  cellRightContainer: {
    flex: 1,
    flexDirection: 'column',
    margin: 10
  },
  recommendHeader: {
    height: 35,
    justifyContent: 'center',
    borderWidth: 1,
    borderColor: '#e9e9e9',
    paddingVertical: 8,
    paddingLeft: 20,
    backgroundColor: 'white'
  },
  itemCellView: {
    flex: 1,
    backgroundColor: '#FFFFFF',
    marginLeft: 10
  },
  thumbnail: {
    width: 80,
    height: 60,
    borderWidth: 1,
    borderColor: '#e9e9e9',
  },
  title: {
    fontSize: 18,
    color: '#999999',
    marginBottom: 8,
    textAlign: 'left'
  },
  id: {
    fontSize: 16,
    textAlign: 'left'
  }
})

```

```

    },
    itemDivice: {
      backgroundColor: '#e9e9e9',
      height: 1,
      flex: 1
    }
  });
  APPRegistry.registerComponent('Demo', () => Demo);

```

如果要展现的列表会很长，或者列表子视图比较复杂，诸如下拉更多等操作，这时候你可能需要关注对 ListView 的优化，诸如复用等性能优化。当然，如果只是简单的展示，系统的 ListView 控件完全可以胜任。

对于 ListView 的优化，建议使用第三方控件 react-native-sglistview，官方地址：<https://github.com/sghiassy/react-native-sglistview>。

4.2.3 Navigator组件

对于 APP 而言，一款完整的应用往往涉及很多的页面，而对于页面之间的跳转，Android 和 iOS 中的实现也各不相同。在 iOS 上，系统通过提供 UINavigationController 控件专门控制页面的跳转。iOS 的实现思路很清晰，开发者只需要为按钮添加 action 事件，系统收到点击之后即可跳转到指定的页面。例如：

```

// 定义一个 Button，点击后跳转到另一个页面
UIButton * button=[UIButton buttonWithType:UIButtonTypeSystem];
button.frame=CGRectMake(130, 220, 100, 30);
[button addTarget:self action:@selector(toNext)forControlEvents:UICtrl
lEventTouchUpInside];
[button setTitle:@"跳转登录" forState:UIControlStateNormal];
[self.view addSubview:button];
// 省略...
-(void)toNext{
  UIBarButtonItem * back=[[UIBarButtonItem alloc]init];
  back.title = @"返回 ";
  self.navigationItem.backBarButtonItem = back;

  SecondViewController * second = [[SecondViewController alloc]init];
  [self.navigationController pushViewController:second animated:YES];
}

```

在 Android 系统中，页面之间的跳转是由工作栈维护的，即通常所说的压栈和弹栈。进入一个页面，一般是入栈，而退出一个页面，就意味着出栈。在 APP 开发中，之所以没有感觉到出栈入栈，是因为 Android 将这些都封装好了，当然开发人员也可以自己维护页面堆栈。

在 Android 开发中，页面跳转一般会用到 Intent 来申明页面跳转。而返回的时候可以通过操作界面所在的 Activity 达到操作堆栈的目的。例如：

```
//A 页面跳转到 B 页面
```

```
Intent intent = new Intent(this, B.class);
context.startActivity(intent);
```

在 React Native 中, 系统为开发者提供了两个组件来实现页面导航: Navigator 和 NavigatorIOS。Navigator 可以实现在 iOS 和 Android 同时使用, 而 NavigatorIOS 则是包装了 UIKit 库的导航功能, 方便用户左划屏幕来返回到上一界面。具体来说, Navigator 是使用 JavaScript 实现的导航组件, 可以同时 in Android 和 iOS 平台使用。而 NavigatorIOS 只能作用于 iOS 平台, 所以笔者建议使用 Navigator 来做页面导航, 如图 4-8 所示。

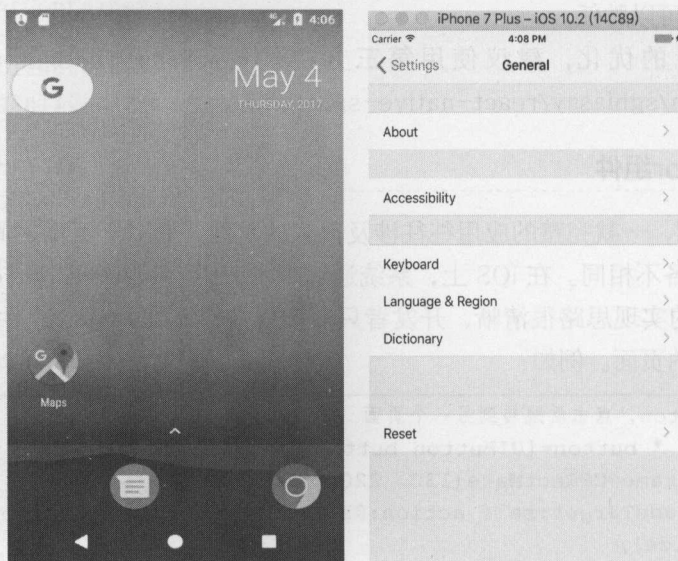


图4-8 Android (物理键) 和iOS的导航栏

Navigator

通过 Navigator (导航器) 让应用可以在不同页面之间跳转。Navigator 用来建立一个路由栈, 用来实现路由状态压入和弹出操作。

在使用路由之前, 要实现页面间的跳转, 需要在 renderScene 方法里面设置需要渲染的场景 (即跳转的页面), 并对路由进行其他配置。相关代码如下。

```
// 渲染场景
```

```
renderScene=(route, navigator) => {
  let Component = route.component;
  return <Component {...route.params} navigator={navigator} />
}
```

本方法主要提供两个参数: route 和 navigator。其中 route 就是路由的页面, navigator 是 Navigator 对象, Navigator 可以提供 pop、push 和 jump 等方法。

使用导航器，首先需要初始化一个路由（initialRoute），该方法如下：

```
// 初始化路由
initialRoute={{ name: defaultName, component: defaultComponent }};
```

再回头看 renderScene 方法，route 参数就是我们通过 initialRoute 初始化的对象，我们通过 route.component 即可获得需要渲染（即跳转到）的对象，然后在返回这个对象时，完成页面的跳转。为了防止出现问题，我们加上 route.component 为空的判断，如果不为空，再进行渲染。

```
if(route.component){
  return <Component {...route.params} navigator={navigator} />
}
```

在 Navigator 中还有一个方法需要注意——configureScene。该方法用来设置指定路由对象的配置信息，例如改变场景的动画或者手势。例如跳转动画：

```
// 跳转动画
iconfigureScene={(route) => {
  return Navigator.SceneConfigs.VerticalDownSwipeJump;
}}
```

Navigator.SceneConfigs 为开发者提供了大量的跳转动画，具体可以参考源码，文件所在目录为：···\node_modules\React-native\Libraries\CustomComponents\Navigator\NavigatorSceneConfigs.JS。

这样，我们就完成了在程序默认页面到自定义页面之间的跳转过程。完整代码如下：

```
import FirstPageComponent from './src/FirstPageComponent';
// 省略...
render() {
  var defaultName = 'FirstPageComponent';
  var defaultComponent = FirstPageComponent;
  return (
    <Navigator
      // 指定了默认的页面
      initialRoute={{ name: defaultName, component: defaultComponent }}
      configureScene={(route) => {
        // 跳转动画
        return Navigator.SceneConfigs.FadeAndroid;
      }}
      renderScene={(route, navigator) => {
        let Component = route.component;
        if(route.component){
          return <Component navigator={navigator} />
        }
      }} />
  )
}
```

```
);
}
```

其中，FirstPageComponent 是默认界面。当然，也可以在这个界面实现一个点击跳转的功能。例如，实现点击一个文本跳转到另一个新的界面。那么，先定义一个文本，然后给文本绑定跳转事件即可。

```
<TouchableOpacity onPress={this.openPage.bind(this)}>
  <Text style={styles.textStyle}> 点我跳转第二个页面 </Text>
</TouchableOpacity>
```

点击后跳转的事件，使用 navigator.push 即可实现：

```
openPage() {
  this.props.navigator.push({
    component: ProductDetail
  })
}
```

除了上面的应用使用到的属性外，Navigator 还提供了很多其他有用的属性。

Navigator 属性和方法

- getCurrentRoutes()

获取当前栈里的路由。
- jumpBack()

跳回之前的路由。
- jumpTo(route)

跳转到已有的场景并且不销毁页面。
- push(route)

跳转到新的场景，并且将场景入栈。
- pop()

返回上一个页面并且对页面进行弹栈操作。
- replace(route)

用一个新的路由替换掉当前场景。
- replaceAtIndex(route, index)

替换掉栈指定序列的路由场景。
- replacePrevious(route):

替换掉之前的场景。
- resetTo(route)

跳转到新的场景，并且重置整个路由栈。
- immediatelyResetRouteStack(routeStack)

用新的路由数组来重置路由栈。

- `popToRoute(route)`
pop到路由指定的场景。在整个路由栈中，处于指定场景之后的场景将会被销毁。
- `popToTop()`
pop到栈中的第一个场景，弹出栈所有的其他场景。

Navigator参数传递

当两个页面相互跳转的时候，免不了需要进行参数的传递，那么，怎么将参数的内容传递到另一个页面呢？

在 Navigator 导航器实例中，一般涉及 3 个对象：FirstPageComponent、SecondPageComponent 和 Navigator 导航器。Navigator 导航器初始化路由的时候获取 FirstPageComponent 作为应用的启动文件。相关代码如下：

```
renderScene=(route, navigator) => {
  let Component = route.component;
  if(route.component){
    return <Component navigator={navigator} />
  }
}
```

当默认的页面通过 navigator 加载到界面上后，怎么通过 Navigator 跳转到第二个页面呢？通过 navigator 的 push 函数，开发者可以很方便的实现页面的跳转。如果需要在跳转过程中传递参数，可以使用 params 来包裹要传递的内容。例如：

```
// 配置第二个页面并传值
if(navigator) {
  navigator.push({
    name: 'SecondPageComponent',
    component: SecondPageComponent,
    params:{
      message:"I am from FirstPageComponent",
    }
  })
}
```

然后第二个界面中，使用 navigator 获取传递的对象即可，这部分的内容是固定的。

```
renderScene=(route, navigator) => {
  let Component = route.component;
  if(route.component){
    return <Component {...route.params} navigator={navigator} />
  }
}
```

至此，使用 Navigator 进行页面间的跳转就完成了，如果页面跳转涉及参数传值，可以在

第二个页面中通过 `props.message` 获取到传递的信息。当然，当第二个页面挂载时，还可以通过 `componentDidMount` 方法获取保存在 `state` 的内容。

```
componentDidMount() {
  this.setState({
    message: this.props.message,
  });
}
```

有时候，需要在返回的时候传递参数内容给上一个页面，也可以使用 `Navigator`。

第三方库

使用 `React Native` 提供的组件虽然能实现页面导航的效果，不过其使用比较复杂，代码结构混乱，不利于代码开发和维护。笔者推荐使用第三方库，截至目前，比较成熟的第三方库有：`react native simple router`、`react-native-router-flux`、`react-navigation` 等。推荐使用 `react-navigation`。

4.2.4 WebView组件

在原生 APP 中，`WebView` 是一个非常实用的组件，它内嵌在原生 APP 中，常用来做一些临时性的活动。这种开发模式即大家熟知的 Hybrid 开发模式，现在依然有很多 APP 在使用这种开发模式。在 iOS 平台使用 `WKWebView` 加载 H5 页面，Android 平台则通过 `WebView` 组件来加载，它们对 H5 页面的支持都比较强大。而在 `React Native` 开发中，系统也为开发者提供了 `WebView` 组件来加载网页内容，使用上也比较简单。使用 `WebView` 组件来加载百度首页的代码如下：

```
const url='http://www.baidu.com/';
// 省略...
<WebView style={styles.container} source={{uri: url}}
// 省略...
/>
```

WebView属性及方法

和原生浏览器组件一样，`Navigator` 内置了大量的属性和方法，如下所示。

- `automaticallyAdjustContentInsets`
是否自动调整内容。
- `contentInset`
设置内容尺寸大小，常见格式：`{top:number,left:number,bottom:number, right:number}`
- `html`
`WebView`加载的HTML文本字符串。
- `injectJavaScript()`
在网页加载前注入一段JS代码。

- `onError()`
在网页加载失败的时候调用。
- `onLoadEnd()`
当网页加载结束时调用，不管加载成功还是失败。
- `onLoadStart()`
当网页开始加载的时候调用。
- `onNavigationStateChange()`
当导航状态发生变化时调用。
- `renderError()`
渲染一个用来显示错误信息的View视图。
- `renderLoagin()`
渲染一个用来显示加载进度指示器的View视图。
- `allowsInlineMediaPlayback(iOS)`
适用于iOS平台。设置使用HTML5播放视频的时候，是在当前页面位置播放还是使用原生的全屏播放器播放。默认值false。
- `bounces(iOS)`
适用于iOS平台。设置是否有界面反弹特性。
- `onShouldStartLoadWithRequest(iOS)`
适用于iOS平台。用来设置拦截WebView加载的URL地址，并根据返回的状态决定是否继续加载。
- `scalesPageToFit(iOS)`
适用于iOS平台。用来设置网页是否自适应缩放到整个屏幕视图以及用户是否可以缩放页面。
- `scrollEnabled(iOS)`
适用于iOS平台。用于设置是否开启页面滚动。
- `domStorageEnabled(Android)`
适用于Android平台。用于控制是否开启DOM Storage。
- `javaScriptEnabled(Android)`
适用于Android平台。用来设置是否开启JavaScript，iOS中的WebView是默认开启的。

WebView实例

为了方便讲解，本例使用 WebView 组件加载百度首页。然后，再看看使用 WebView 组件加载本地 HTML 文件实例。使用 WebView 加载百度首页的代码如下：

```
const url='http://www.baidu.com/';
const {width, height} = Dimensions.get('Window');
// 省略...
<WebView
```



```

style={{width:width,height:height-20,backgroundColor:'gray'}}
source={{uri:url}}
javaScriptEnabled={true}
domStorageEnabled={true}
scalesPageToFit={false}
/>

```

运行效果如图 4-9 所示。



图4-9 WebView加载百度首页

要使用 WebView 组件加载本地 HTML，首先，需要区分是开发还是发布环境，还要区分是 Android 平台还是 iOS 平台。对于不带 query 参数的可以使用下面的方法。

```

let source;
if (develop) {
  source = reqUIre('./index.html');
} else {
  source = Platform.OS === 'ios' ? reqUIre('./index.html') : { uri: 'file:///
android_asset/index.html' };
}
// 省略...
<WebView source={source} {...props} />

```

如果需要在 uri 后面加 query 参数，例如 path/to/file.html?page=1，一般采用如下方式：

```

let source;
const _source = resolveAssetSource(reqUIre('./index.html'));
if (develop) {

```

```

    source = { uri: '${_source.uri}&id=${article.id}' };
  } else {
    const sourceAndroid = { uri: 'file:///android_asset/index.html?id= ${article.id}' };
    const sourceIOS = { uri: 'file:///${_source.uri}?id=${article.id}' };
    source = Platform.OS === 'ios' ? sourceIOS : sourceAndroid;
  }
  // 省略...
<WebView source={source} {...props} />

```

关于 WebView 组件的使用就讲到这里。其实 React Native 为我们提供的组件还有很多，例如 TextInput（输入框）、Switch（开关按钮）、ScrollView（滚动组件）等。

4.3 平台特定组件

对于 Android 和 iOS 平台来说，不是所有的 React Native 组件都是通用的，同样，也不是所有的交互方式在 Android 和 iOS 都适合，这就意味着开发者需要在应用开发中使用平台定制的组件。值得注意的是，在 React Native 系统提供的平台组件中，为了区分不同的平台，React Native 系统分别以 iOS 或 Android 后缀来标识平台特定组件。

既然如此，那么我们怎么处理跨平台应用特定组件的属性呢？还记得我们新建项目的时候系统默认新建的 index.android.js 和 index.ios.js 这两个文件吗？在 React Native 中，针对跨平台的元素，都可以通过命名来对 Android 和 iOS 进行不同的实现。

这些特点的平台组件中，常见的有 <TabBarIOS>、<ToolbarAndroid>、<DatePickerIOS> 和 <DatePickerAndroid>、<ProgressBarAndroid> 和 <ProgressViewIOS> 等。

4.3.1 TabBarIOS和TabBarIOS.Item组件

目前大多数 APP 都是按照 Tab 页面进行模块间切换的，这类 APP 包括微信、微博、淘宝等。在原生 iOS 开发中，我们可以使用 UITabBarController 来实现 Tab 切换的效果。在 React Native 中，我们可以通过 TabBarIOS 和 TabBarIOS.Item 组件来实现选项卡切换效果，当然，这两个组件只支持 iOS 平台。

TabBarIOS属性及方法

TabBarIOS 常用的属性如下：

- 继承了 View 的所有属性。
- barTintColor
Tab 栏的背景颜色。
- tintColor
当前被选中标签的图标颜色。
- translucent
设置 Tab 栏是不是半透明效果。

TabBarIOS.Item属性及方法

TabBarIOS.Item 常用的属性如下。

- 继承View的所有属性。
- style
设置样式风格，继承View的所有样式风格。
- title
Tab选项卡下面的文字信息。如果你设置了SystemIcon属性，那么该属性会被忽略。
- badge
在图标右上方显示的角标。
- icon
给当前标签设置自定义图标。
- onPress
当Tab按钮被选中的时候进行回调，你可以通过selected={true}来设置组件被选中。
- selected
决定子视图是否可见，如果可见，则可看到选中的标签页面。
- selectedIcon
当Tab按钮被选中的时候显示的自定义图标。如果定义了icon属性，但是当前的selectedIcon属性没有设置，那么该图标会被设置成蓝色。
- systemIcon
系统预定义的图标，是一个枚举类型函数，主要的默认系统图标有：bookmarks、contacts、downloads、favorites、featured、history、more、most-recent、most-viewed、recents、search、top-rated。

使用案例

TabBarIOS 的使用非常简单。不过需要注意的是，在使用过程上，必须给 TabBarIOS 设置尺寸，不然可能会造成界面无法显示的问题，在使用 TabBarIOS 实现 Tab 切换功能时，需要配合 TabBarIOS.Item 一起使用。

```
// 导包
import { TabBarIOS } from 'react-native';
// 省略...
render() {
  return (
    <View style={styles.container}>
      <TabBarIOS
        style={{height:49, width: width}}
      >
      </TabBarIOS>
    </View>
  );
}
```

```
);
}
```

接着需要使用 `<TabBarIOS.Item>` 组件来添加 Item 子项，TabBarIOS 最多只能包含 5 个 Item 子项。

```
<TabBarIOS.Item
  systemIcon="bookmarks" // 系统图标 (bookmarks)>
</TabBarIOS.Item>
```

例如，在下面的示例项目中，使用 TabBarIOS 实现了 Tab 的切换效果，如图 4-10 所示。

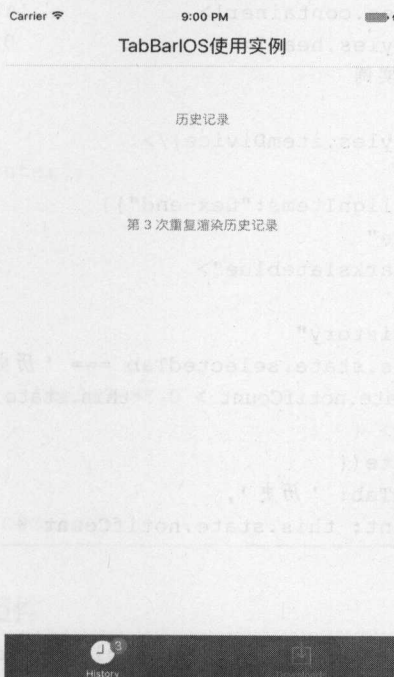


图4-10 使用TabBarIOS实现Tab切换

在本例中，使用系统提供的图标来完成了 Tab 切换的效果，并对 TabBarIOS 和 TabBarIOS.Item 提供的属性进行应用。核心代码如下：

```
class TabBarIOSScene extends Component {
  constructor(props) {
    super(props);
    this.state={
      selectedTab: '历史',
      notifCount: 0,
      presses: 0,
    };
  }
}
```



```

renderContent(pageText: string, num?: number) {
  return (
    <View style={styles.tabContent}>
      <Text style={styles.tabText}>{pageText}</Text>
      <Text style={styles.tabText}>第 {num} 次重复渲染 {pageText}</Text>
    </View>
  );
}

render() {
  return (
    <View style={styles.container}>
      <Text style={styles.head}>
        TabBarIOS 使用实例
      </Text>
      <Text style={styles.itemDivice}/>
      <TabBarIOS
        style={{flex:1,alignItems:"flex-end"}}
        tint="white"
        barTintColor="darkslateblue">
        <TabBarIOS.Item
          systemIcon="history"
          selected={this.state.selectedTab === '历史'}
          badge={this.state.notifCount > 0 ? this.state.notifCount : undefined}
          onPress={() => {
            this.setState({
              selectedTab: '历史',
              notifCount: this.state.notifCount + 1,
            });
          }}
        >
          {this.renderContent('历史记录', this.state.notifCount)}
        </TabBarIOS.Item>
        <TabBarIOS.Item
          systemIcon="downloads"
          selected={this.state.selectedTab === '下载'}
          onPress={() => {
            this.setState({
              selectedTab: '下载',
              pressesCount: this.state.presses + 1
            });
          }}
        >
          {this.renderContent('下载页面', this.state.pressesCount)}
        </TabBarIOS.Item>
      </TabBarIOS>
    </View>
  );
}

```

```

    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20
  },
  head: {
    fontSize: 20,
    textAlign: 'center',
    paddingVertical: 10
  },
  tabContent: {
    flex: 1,
    alignItems: 'center',
  },
  tabText: {
    color: 'gray',
    margin: 50,
  },
  itemDivice: {
    backgroundColor: '#e9e9e9',
    height: 1,
  }
});
export default TabBarIOSScene;

```

4.3.2 ToolbarAndroid组件

ToolbarAndroid，是仅作用于 Android 平台的工具栏组件。一个 Toolbar 组件就是一个提供导航的工具栏，在这个工具类视图中包含导航图标（返回按钮）、标题和功能按钮，如图 4-11 所示。因为此组件只针对 Android 平台，所以开发完成后，只能运行在 Android 系统平台上。

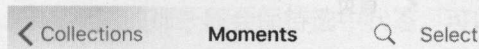


图4-11 Toolbar使用

ToolbarAndroid属性及方法

ToolbarAndroid 常用的属性如下。

- actions

设置功能菜单中的可用功能，值为数组，这个属性和跟Android原生的toolbar非常类似。本属性用来显示右侧功能区域的图标或文字，如果显示不下，会被放进一个弹出菜单中，

以列表形式显示。

- title
工具栏的标题。
- titleColor
标题文字颜色。
- subtitle
工具栏的副标题。
- subtitleColor
工具栏的副标题文字颜色。
- logo
设置工具栏的logo。
- icon
设置功能图标，例如require('./some_icon')。
- navIcon
设置工具栏的导航图标。
- overflowIcon
设置功能列表的弹出菜单的图标。
- show
直接作为icon显示还是隐藏后在弹出菜单里显示。
- onActionSelected()
当功能列表中某个功能被选中的时候调用此回调。传递给此回调的唯一参数作为功能在actions数组中的位置。
- onIconClicked()
当图标被点击时，回调此函数。

ToolbarAndroid示例

只有 logo 和 title 的 ToolbarAndroid 视图，如图 4-12 所示。



图4-12 ToolbarAndroid属性

```
<ToolbarAndroid
  style={styles.toolbar}
  logo={require('./image/back_icon.png')}
  title=" 首页 "/>
```

使用 navIcon 和 action 的 ToolbarAndroid 视图，如图 4-13 所示。

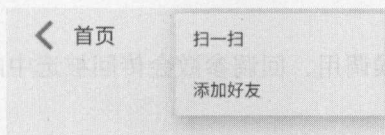


图4-13 ToolbarAndroid属性

```
<ToolbarAndroid
  style={styles.toolbar}
  logo={require('./image/back_icon.png')}
  title=" 首页 "
  actions={[{title: ' 扫一扫 ', show: 'never'}, {title: ' 添加好友 ', show: 'never'}]}
  onActionSelected={this.onActionSelected}/>
```

除了这些最基本的用法之外，在具体项目中，还可以配合其他组件来实现诸如跳转的功能。关于 Navigator 跳转功能的实现，大家可以参考之前章节的讲解。

4.3.3 SegmentedControlIOS组件

SegmentedControlIOS，是一个仅限于 iOS 平台的分段组件。它对应于原生 iOS 系统中的 UISegmentedControl 控件，使用该组件可以实现点击不同的分段以切换不同的视图的效果。在 React Native 中，系统为开发者提供了 SegmentedControlIOS 组件用以实现 UISegmentedControl 分段显示的效果，如图 4-14 所示。

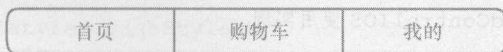


图4-14 SegmentedControlIOS实现分段效果

SegmentedControlIOS属性及方法

SegmentedControlIOS 主要有以下属性需要掌握。

- 继承 View 的所有属性和样式。
- enabled
设置控件是否可用。默认为 true。
- momentary
该值为 true 的时候，分段视图的那一段会保持选中状态。可以通过函数 onChange 进行分段视图状态的回调。
- selectedIndex
被选中的分段下标。
- tintColor
被选中的分段的颜色
- values[数组]
分段标题的集合，按照索引的顺序进行排列。

- onChange()
当用户点击某一段的时候调用，回调参数会传回被选中的分段。
- onChangeValueChange()
当用户点击某一分段的时候调用。

SegmentedControlIOS示例

SegmentedControlIOS 作为 iOS 设备上的分段选项组件，用来进行不同项目视图的点击切换，如图 4-15 所示。在 Android 平台，要想实现相同的效果，我们需要自定义组件来实现。



图4-15 SegmentedControlIOS分段切换

```
class SegmentedControlIOSView extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.text}>
          SegmentedControlIOS 使用实例
        </Text>
        <SegmentedControlIOS
          style={styles.segmentedStyle}
          values={['Android', 'iOS', 'Java', 'React']}
          tintColor='gray'
          selectedIndex={1}
          onChange={(value) => alert('选中了: '+value)} />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20,
  },
  segmentedStyle: {
    marginTop: 20,
    margin: 10,
    height: 30,
  },
});
```

```

width:300,
alignSelf:'center'
},
text: {
  alignSelf:'center',
  marginTop:20,
},
});

```

```
export default SegmentedControlIOSView;
```

4.3.4 ViewPagerAndroid组件

在原生 Android 开发中，ViewPager 是比较常见的页面切换控件。而在 React Native 中，系统为开发者提供了一个名为 ViewPagerAndroid 的组件来实现 ViewPager 的效果。在 React Native 的官方项目 UIExplorerAPP 中也可以看到 ViewPagerAndroid 的身影。

每一个 ViewPagerAndroid 的子容器会被视为一个单独的页面。不过需要注意的是，所有的子视图都必须是纯 View，而不能是自定义的复合容器。在使用上 ViewPagerAndroid 也是非常的简单，只需要将子容器添加到 ViewPagerAndroid 容器下即可。示例代码如下：

```

<ViewPagerAndroid
  style={styles.viewPager}
  initialPage={0}>
  <View style={styles.pageStyle}>
    <Text> 第一个页面 </Text>
  </View>
  <View style={styles.pageStyle}>
    <Text> 第二个页面 </Text>
  </View>
  // 省略...
</ViewPagerAndroid>

```

ViewPagerAndroid属性及方法

ViewPagerAndroid 常用属性如下。

- 继承View的全部属性。
- initialPage

初始索引页（默认为0），可以通过在onPageSelected监听页面的变化，进而使用 setPage方法更新页面。

- onPageScroll

页面滑动的时候执行。该属性会回调传入的event.nativeEvent对象的具体信息（position和offset），其中，position为当前页面的索引，offset为滑动所占的百分比。

- onPageScrollStateChanged

页面滑动的状态，主要有3个状态：idle（空闲）、dragging（拖动中）、settling（手松开处理中）。

- scrollEnabled

该属性用来设置是否可以滑动。默认为true。

- onPageSelected()

该方法用于拖拽滑动切换完成之后的回调函数。该方法的回调参数中的event.nativeEvent对象会携带一个position的参数。

ViewPagerAndroid示例

如图4-16所示，使用ViewPagerAndroid组件可以很轻松地实现图片轮播效果，当然，我们还可以通过对onPageScroll、onPageScrollStateChanged、onPageSelected等函数的监听来实现具体的业务开发，通过调用setState函数还可以更改ViewPagerAndroid的属性状态。示例代码如下：



图4-16 ViewPagerAndroid图片切换

```
class ViewPagerAndroidView extends Component {
  constructor() {
    super();
    this.state={
      tabIndex:0,
    }
  }
}
```

// 页面切换时执行

```
onPageScroll=(event)=>{
  this.setState({pagePosition:event.nativeEvent.position,
    offset:event.nativeEvent.offset});
}
```

```
render() {
  return (
```

```

<View style={styles.container}>
  <ViewPagerAndroid
    onPageScroll={this.onPageScroll}
    style={{height:200,width:500}} >
    <View>
      <Image
        style={{height:200,width:500}} source={{uri:'http://pl.so.
qhimgsl.com/dmfd/326_204_/t014a9280f55313598d.jpg'}}/>
      </View>
      <View>
        <Image
          style={{height:200,width:500}}
          source={{uri:'http://pl.so.qhimgsl.com/sdr/1365_768_/
t013a5b13b26f53e9a1.jpg'}}/>
        </View>
        <View>
          <Image
            style={{height:200,width:500}}
            source={{uri:'http://p4.so.qhimgsl.com/sdr/1365_768_/
t0175e724d3dc217114.jpg'}}/>
          </View>
        </View>
        <Text style={styles.text}>当前第 {this.state.pagePosition} 页</Text>
      </View>
    );
  }
}

```

除了实现上面的效果外，开发者还可以使用 ViewPagerAndroid 来实现诸如引导图、图片轮播等效果。

4.4 Touchable系列组件

前面在讲 React Native 基础的时候，提到了 React Native 的手势与触摸响应系统，其中最核心的就是 Touchable 系列组件。之所以将 Touchable 单独提出来，是因为 Touchable 由一系列组件构成，且具有很强的代表性，该系列组件主要包含以下组件：TouchableWithoutFeedback、TouchableHighlight、TouchableNativeFeedback、TouchableOpacity。其中 TouchableWithoutFeedback 是不带反馈效果的，其他三个都是带有触摸反馈效果的，可以理解为他们三个都是通过继承自 TouchableWithoutFeedback 扩展而来。其主要区别和联系如下：

- TouchableHighlight：高亮触摸。用户点击时，会产生高亮效果。
- TouchableOpacity：透明触摸。用户点击时，点击的组件会出现透明过渡效果。
- TouchableNativeFeedback：用于封装视图，使其可以正确响应触摸操作（仅限 Android 平台）。

4.4.1 TouchableWithoutFeedback

React Native 官方文档中特别强调，除非你不得不用，否则不要用这个组件。这个组件不提供任何视觉上的反应效果。并且，TouchableWithFeedback 只支持一个字节节点，如果你需要设置多个子视图组件，那么就需要使用 View 节点进行包装。

TouchableWithoutFeedback属性及方法

TouchableWithoutFeedback 提供的属性和方法如下。

- accessibilityComponentType
设置可访问的组件类型。
- accessibilityTraits
设置可访问组件的特征。
- accessible
设置当前组件是否可以访问。
- delayLongPress
设置从onPressIn方法开始，到onLongPress被调用的这一段时间。
- delayPressIn
设置从用户触摸控件开始，到onPressIn被调用的这一段时间。
- delayPressOut
设置从用户触摸事件释放开始，到onPressOut被调用的这一段时间。
- onLayout()
当组件加载或者该组件的布局发生变化的时候调用。
- onLongPress()
当用户长时间按压组件（长按效果）的时候调用。
- onPressIn()
当用户开始触摸组件的时候调用。
- onPressOut()
当用户触摸结束的时候调用。
- pressRetentionOffset

在当前视图被禁用的前提下使用这个属性，本属性决定当手指移开多远距离之后，会不再激活按钮，如果手指再次移回范围内，按钮会被再次激活。

在实际项目开发中，该组件的使用频率比较低，更多的时候使用其他三种组件替代。

4.4.2 TouchableHighlight

有些时候，当用户按下屏幕后希望系统给出一些视觉上的提示，而 TouchableHighlight 组件就提供了相应的功能。当用户触摸屏幕时，该视图的不透明度会降低，同时会呈现相应的颜色变化（视图变暗或者变亮）。如果去查看该组件的源代码就会发现，当屏幕被按下的时候，该

底层添加了一个新的视图。所以，TouchableHighlight 只能进行一层嵌套，不能多层嵌套，如果需要多层嵌套，就需要使用 View 节点进行包装。

TouchableHighlight属性

TouchableHighlight 组件提供的属性如下。

- style
设置控件的显示风格。
- underlayColor
触摸或者点击控件的时候显示出的颜色。
- activeOpacity
设置组件在进行触摸的时候的不透明度（取值在0与1之间）。
- onHideUnderlay()
当底层被隐藏的时候调用。
- onShowUnderlay()
当底层显示的时候调用。

TouchableHighlight示例

使用 TouchableHighlight 点击改变颜色的示例界面如图 4-17 所示。

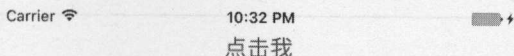


图4-17 TouchableHighlight点击改变颜色

```
class TouchableHighlightView extends Component {
  render() {
    return (
      <View style={styles.container}>
        <TouchableHighlight activeopacity='0.5' underlaycolor='red'>
          <Text style={styles.textStyle}> 点击我 </Text>
        </TouchableHighlight>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 25,
    alignItems: 'center',
  },
  textStyle: {
    fontSize: 18,
  }
});
```

```

        color:'#434343'
    },
  });

```

4.4.3 TouchableOpacity

有的时候，用户希望触摸屏幕后屏幕其透明度会发生变化，TouchableOpacity 就可以帮助我们实现这种效果。当按下的时候，视图的不透明度会降低。但是，这一过程并不会真正改变视图层级，也不会带来一些奇怪的副作用。

TouchableOpacity属性

TouchableOpacity 的属性比较简单。

- activeOpacity

设置当用户触摸的时候，组件的透明度的变化（取值在0~1之间）。

TouchableOpacity示例

TouchableOpacity 使用上也比较简单，不再详述。

```

<TouchableOpacity activeOpacity={0.9}>
  <Text> 点击查看效果 </Text>
</TouchableOpacity>

```

4.4.4 TouchableNativeFeedback

该组件是 Android 特有，在 Android 平台使用该组件可以显示触摸状态的变化。需要注意的是，该组件只支持仅有的一个 View 作为子节点。在底层技术实现上，TouchableNativeFeedback 会创建一个新的 RCTView 节点替换当前的子 View，并附带一些额外的属性。

TouchableNativeFeedback属性和方法

- 本组件继承了所有 TouchableWithoutFeedback 的属性。
- background

设置背景资源的类型，该属性会传入 type 属性和基于 type 属性的额外资源对象。官方推荐采用如下的静态方法来进行生成该 dictionary 对象。
- TouchableNativeFeedback.SelectableBackground(): 本属性会创建一个 android 默认主题背景 (?android:attr/selectableItemBackground)。
- TouchableNativeFeedback.SelectableBackgroundBorderless(): 本属性会创建无框的主题背景 (?android:attr/selectableItemBackgroundBorderless)。该类型只对 Android API level 21+ 适用。
- TouchableNativeFeedback.Ripple(color, borderless): 本属性会创建组件被按下时的水滴效果。可以通过 color 参数来指定颜色，如果参数 borderless 是 true，那么

涟漪还会渲染到视图的范围之外。这个类型只对Android API level 21+适用。

TouchableNativeFeedback示例

```
<TouchableNativeFeedback
  onPress={this.onPressButton}
  background={TouchableNativeFeedback.SelectableBackground()}>
  <View style={{width: 150, height: 100, backgroundColor: 'red'}}>
    <Text style={{margin: 30}}>Button</Text>
  </View>
</TouchableNativeFeedback>
```

4.5 小结

在学习 React Native 的过程中，读者需要明白，组件并不是真正的 DOM 节点，而是存在于内存中的一种数据结构，叫做虚拟 DOM，只有当插入真正的元素之后，才会成为真正的 DOM。如果想要通过组件获取真正的 DOM 节点，可以通过 ref 属性设置标记，并通过 this.refs 来访问这个组件。

在本章中，我们主要学习了 React Native 的基础组件，以及针对 iOS、Android 平台的平台特定组件，并重点介绍了 Touchable 系列组件。通过组件的学习，可以让我们更加高效地开发出复杂的页面。

常用API介绍

在 React Native 页面开发过程中，只有配合组件和常用的 API，才能更好地开发出合格的移动产品。在前面章书中，已经提到了一些 API，这些 API 构成了 React Native 页面的基本元素。在这一章，将重点介绍一些常用的 API，并通过实例讲解这些 API 在实际开发中的用途。

5.1 AppRegistry

在 React Native 开发中，AppRegistry 模式是最基本的模块，是 React Native 应用的 JavaScript 运行入口。具体应用时，使用 AppRegistry.registerComponent 进行系统注册，原生系统通过加载运行 bundle 文件包，系统经过内部解析，最后调用 AppRegistry.runApplication 来运行 React Native 程序。当页面销毁的时候，通过调用 AppRegistry.unmountApplictionComponentAtRootTag 函数结束应用。需要注意的是，unmountApplictionComponentAtRootTag 的参数要和 runAPPLICATION 的参数保持一致。

AppRegistry 应当在其他模块导入之前尽可能早点被导入，以确保 JavaScript 运行环境在其他模块之前被准备好。

AppRegistry 属性和方法

AppRegistry 常见的属性和方法如下。

- registerConfig(config:Array)
静态方法，注册配置。
- registerComponent(AppKey:string, getComponentFunc: ComponentProvider)

注册入口组件。

- registerRunnable(AppKey:string, func :Function)

注册进程。

- registerAppKeys()

获取所有组件的keys值。

- runApplication(AppKey:string, AppParameters:any)

启动应用。

- unmountApplicationComponentAtRootTag()

结束应用。

AppRegistry实例

在 React Native 项目中，使用 AppRegistry.registerComponent 来注册应用程序：

```
AppRegistry.registerComponent('HelloWorld', () => HelloWorld);
```

在 Xcode 或者 Android Studio 中，我们可以看到应用的一些启动信息，这些信息由 runApplication 打印。我们还可以使用 registerRunnable 来注册 AppKey。

例如，通过 runApplication 获取应用运行信息，效果如图 5-1 所示。

```
alert(AppRegistry.getAppKeys());
```

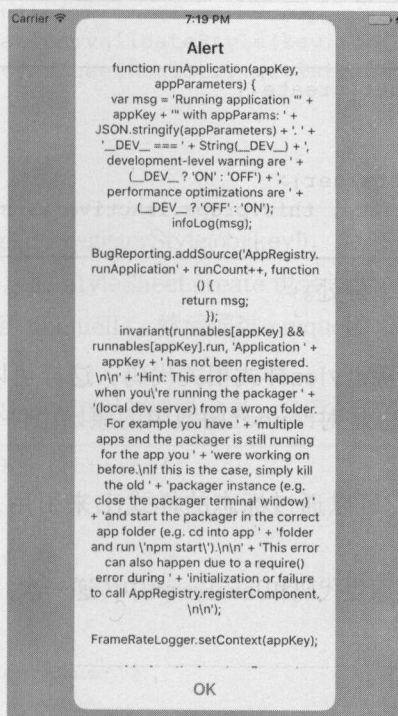


图5-1 使用runApplication获取应用信息

例如,使用 `getAppKeys` 获取 `AppKey` 信息:

```
AppRegistry.registerRunnable('helloWord',function(){
  console.log('hello React Native');
})
alert(AppRegistry.getAppKeys());
```

`AppRegistry` 是 `React Native` 最基本的 API 之一,同时它也是使用频率最高的 API,在以后的项目中会经常用到。

5.2 StyleSheet

在前端开发中,为了达到某种显示效果,往往需要使用 `CSS` 样式表来配合页面标签的使用。而在 `React Native` 中,`StyleSheet` 提供了一种类似 `CSS` 样式表的抽象。在新建项目的时候,在项目的启动页面,我们常常会看到如下的样式申明:

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    borderColor: '#d6d7da',
  },
  // 省略样式
});
```

然后在组件中引入样式文件:

```
const styles = StyleSheet.create({
  // 省略样式
});
<View style={styles.container}>
  <Text style={[styles.title, this.props.isActive && styles.activeTitle]} />
</View>
```

使用 `StyleSheet` 主要有以下好处。

代码层面

- 从 `render` 渲染方法中移除 `styles` 样式相关代码,这样可以使代码更加容易阅读。
- 通过对不同样式命名,也是对 `render` 函数中的原始组件作用的一种标记方式。

性能层面

- 利用 `StyleSheet`,我们可以通过标志的样式 ID 来引用,而不是每次都创建一个新的 `Style` 对象。
- 该样式允许通过桥接,在原生代码和 `JavaScript` 中传递一次,后面全部通过该 ID 进行引用。

StyleSheet属性

`StyleSheet` 提供的属性如下。

- `hairlineWidth`

定义当前平台上的最细的宽度。可以用于边框或是两个元素间的分隔线。

该属性会根据当前平台信息，自动转换成一根很细的线，类似于平台像素。

例如{ borderBottomWidth: StyleSheet.hairlineWidth }。

- flatten

将一个样式对象的数组转换成一个聚合样式对象。同时，该属性还支持通过查找ID，返回一个stylesheet.register。

- absoluteFillObject

你可以使用此属性来创建一个自定义的样式表。例如：

```
const styles = StyleSheet.create({ wrapper: { ...StyleSheet.
absoluteFillObject, top: 10, backgroundColor: 'transparent' }, });
```

StyleSheet方法

- create()

通过给定的对象创建一个StyleSheet样式表。

StyleSheet渲染组件原理

StyleSheet 用来渲染组件的样式，源码如下：

```
const styles = StyleSheet.create({
class StyleSheet {
  static create(obj: {[key: string]: any}): {[key: string]: number} {
    var result = {};
    for (var key in obj) {
      StyleSheetValidation.validateStyle(key, obj);
      result[key] = StyleSheetRegistry.registerStyle(obj[key]);
    }
    return result;
  }
}
```

如上所示，StyleSheetRegistry.registerStyle(obj[key])，会根据 key 返回一个 uniqueID，styles 存储 key 以及对应的 uniqueID。而 StyleSheet.create 也会返回每个 key 对应的 uniqueID，设置组件属性的时候，根据 key 获取到 uniqueID，然后通过 uniqueID 获取对应的样式并更新到界面上。

StyleSheet示例

在实际开发中，使用 StyleSheet 可以创建样式表，然后在组件中引入对应的样式表即可。例如：

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    borderColor: '#d6d7da',
  }
  // 省略样式
});
<View style={styles.container}>
  ...
</View>
```

5.3 AppState

在 React Native 中，核心概念包括组件、属性、状态。状态是 React 的基本属性之一，通过对当前运行状态的监听，从而做出不同的反应。

在 React Native 中，我们可以使用 `AppState.currentState` 来获取应用的状态。应用的状态（`AppState`）主要分为 3 种。

- `active`：前台运行中。
- `background`：后台运行中。
- `inactive`：运行的过渡状态。

AppState属性和方法

`AppState` 提供的常见的属性和方法如下。

- `currentState`

获取应用当前的状态。

- `addEventListener(type: string, handler: Function)`

添加一个函数，监听应用状态的变化。

- `removeEventListener(type: string, handler: Function)`

移除监听函数。

AppState基本用法

使用 `AppState.currentState` 可以获取当前的状态、在启动过程中，获取的 `currentState` 可能为 `null`，这时候不用着急，系统会尝试再次获取，直到 `AppState` 从原生代码得到通知为止。

使用 `AppState` 获取当前应用状态的代码如下：

```
constructor(props) {
  super(props);
  this.state = {
    currentAppState: AppState.currentState,
  };
}

componentDidMount() {
  AppState.addEventListener('change', this.handleAPPStateChange);
}

componentWillUnmount() {
  AppState.removeEventListener('change', this.handleAPPStateChange);
}

handleAppStateChange = (nextAppState) => {
  if (this.state.APPState.match(/inactive|background/) && nextAppState === 'active') {
```

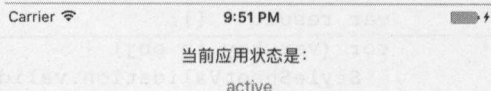


图5-2 使用APPState获取应用状态

```

    console.log('APP has come to the foreground!')
  }
  this.setState({AppState: nextAppState});
}

render() {
  return (
    <View style={styles.container}>
      <Text style={styles.styleText}>
        当前应用状态是:
      </Text>

      <Text style={styles.styleAPPState}>
        {this.state.currentAppState}
      </Text>
    </View>
  );
}

```

使用 `addEventListener` 添加对状态的监听，或者通过 `removeEventListener` 移除监听。例如，我们新增一个内存报警的监听 `memoryWarning`：

```

// 新增内存监听
componentWillMount() {
  AppState.addEventListener('memoryWarning', function(){
    console.log("内存报警...");
  });
}

```

移除相关监听需要调用 `removeEventListener` 函数，示例如下：

```

// 移除监听
componentWillUnmount() {
  AppState.removeEventListener('memoryWarning', this.handleAppStateChange);
}
// 省略...
handleAppStateChange(APPState) {
  console.log('当前状态为:'+APPState);
}

```

5.4 AsyncStorage

AsyncStorage 是一个异步、持久化的、以键值对形式进行数据存储的简单存储系统，它对于 APP 来说是全局性的。它的作用等价于 iOS 系统中的 `NSUserDefaults` 或 Android 系统中的 `SharedPreferences`。AsyncStorage 可用来替换老旧的本地存储方案：`LocalStorage`。

AsyncStorage 是 React Native 提供了一种轻量级数据存储方式，因为使用了 Key-Value 存储系统，所以是不支持 SQL 语句的。

AsyncStorage方法

AsyncStorage 提供的常见方法如下。

- `static getItem(key:string:callback:(error:result))`
根据键来获取值，获取的结果会返回到回调函数中。
- `static setItem(key:string:value:string:callback:(error))`
设置键值对，并将完成后将结果返回给 callback 函数。
- `static removeItem(key:string:callback:(error))`
根据键移出一项数据，并将结果返回给 callback 函数。
- `static mergeItem:(key:string:value:string:callback:(error))`
合并现有的值和新输入的值，并将结果返回给 callback 函数。
- `static clear(callback:(error))`
清除所有的 AsyncStorage 数据，并返回结果。
- `static getAllKeys(callback:(error))`
获取所有的键，并将结果返回给 callback 函数。
- `static multiGet(keys,callback:(errors:result))`
获取keys所包含的所有字段的值，并将结果返回给callback函数，形式为key-value数组形式的数组。
- `static multiSet(keyValuePairs:callback:(errors))`
multiSet和multiMerge都接受一个与multiGet输出值一致的key-value数组，并将结果返回给callback函数。
- `static multiRemove(keys:callback(errors))`
删除所有键在keys数组中的数据，并将结果返回给callback函数。
- `static multiMerge(keyValuePairs:callback:(errors)):`
将多个键值合并，keyValuePairs是字符串中的二维数组，并将结果返回给callback函数。

AsyncStorage示例

作为一个轻量级的数据存取方案，AsyncStorage 可以实现数据的增删改查功能，界面如图 5-3 所示。为了方便介绍，本篇以新增数据为例，来讲解 AsyncStorage 的用法。

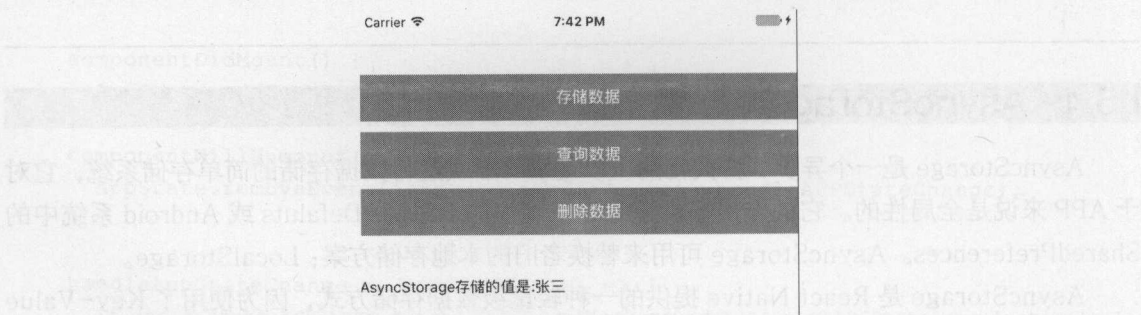


图5-3 使用AsyncStorage实现增删改查操作

下面是使用 AsyncStorage.setItem 来保存数据的例子。

```
saveData(key, vaule, callback) {
  AsyncStorage.setItem(key, vaule, (error, result) => {
    if (error) {
      alert(' 存储失败 ');
    } else {
      this.setState({
        data: vaule
      });
    }
  });
}
```

当然，为了方便，还需要插入和更新方法做一个统一的封装。例如，下面是例用 key 更新数据的示例代码：

```
updateDataByKeys(key, isAdd) {
  AsyncStorage.getItem(this.favoriteKey, (error, result) => {
    if (!error) {
      var favoriteKeys = [];
      if (result) {
        favoriteKeys = JSON.parse(result);
      }
      var index = favoriteKeys.indexOf(key);
      if (isAdd) {
        if (index === -1) favoriteKeys.push(key);
      } else {
        if (index !== -1) favoriteKeys.splice(index, 1);
      }
      AsyncStorage.setItem(this.favoriteKey, JSON.stringify(favoriteKeys));
    }
  });
}
```

除了插入和更新操作之外，AsyncStorage 实现查询和删除功能的核心方法如下：

// 查询数据

```
loadData() {
  thisData = this;
  AsyncStorage.getItem(keyName, function (error, result) {
    if (!error) {
      alert(' 数据查询结果 '+result);
      thisData.setState({
        data: result === null ? ' 数据已经删除 ' : result
      });
    }
  });
}
```

// 删除指定的数据


```

delData(){
  thisData = this;
  AsyncStorage.removeItem(keyName, function (error) {
    if (!error) {
      thisData.setState({
        data: '数据已经删除'
      })
    }
  })
}

```

AsyncStorage 虽然实现了数据的轻量存储功能，但是如果直接使用并不是十分方便，需要在 AsyncStorage 的基础上做一层抽象封装，或者直接使用 React Native 中文网封装维护的 react-native-storage 库。

5.5 PixelRatio

在原生应用开发中，为了满足不同尺寸不同像素手机的显示需求，往往需要做分辨率适配。在分辨率适配上最核心的概念莫过于像素密度和设备像素，而在 React Native 中，PixelRatio 提供了访问设备像素密度的方法。

在 PixelRatio 的使用场景中，最基本的莫过于界面显示上的适配。例如，根据像素密度设置图片大小。一般的做法是，选取一个基础像素尺寸，然后根据设备像素，将图片显示的尺寸乘以像素比。

```

var image = getImage({
  width: PixelRatio.getPixelSizeForLayoutSize(200),
  height: PixelRatio.getPixelSizeForLayoutSize(100),
});
<Image source={image} style={{width: 200, height: 100}} />

```

上面代码的意思是，如果你要在屏幕上显示一个宽 200、高 100 像素的图片，由于设备的分辨率不一样，要想达到 200×100 的显示效果，那么就需要用原像素 × 像素密度，PixelRatio.getPixelSizeForLayoutSize(200) 方法会根据当前设备的 pixelratio 返回对应的结果。例如，当前设备的 pixelratio 为 2，则返回 400 (200×2)，最后生成的参数为 { width: 400,height: 200 }。

pixelratio方法

pixelratio 的常见方法如下。

- get()

返回屏幕的像素密度，如图 5-4 所示。
- PixelRatio.get()==1 mdpi Android设备 (160 dpi)
- PixelRatio.get()==1.5 hdpi Android设备 (240 dpi)
- PixelRatio.get()==2 iPhone4(4S)、iPhone 5(5C、5S)、iPhone 6,xxdpi Android设备 (320 dpi)
- PixelRatio.get()==3 iPhone6 Plus以上,xxhdpi Android设备 (480 dpi)

- `PixelRatio.get()`==3.5 Nexus 6

- `getFontScale()`

获取文字大小的缩放比例，然后针对不同的平台，设置不同的字体大小。

不过需要注意的是，本函数仅仅在Android设备上实

现了，可以通过【Settings】→【Display】→【Font Size】进行设置。而在iOS平台上使用本函数会返回默认的像素密度。

- `getPixelSizeForLayoutSize()`

将一个布局尺寸（dp）转换为像素尺寸（px），并返回一个整数数值。

pixelratio示例

例如，使用 pixelratio 的 `get()` 获取设备的屏幕像素。

```
<Text style={styles.textStyle}>
  当前的屏幕像素密度比例为：{PixelRatio.get()}
</Text>
```



图5-4 获取设备屏幕像素

5.6 Animated

动画是移动开发的重要组成部分，也是提升用户体验非常重要的部分。在 React Native 中，动画相关的 API 叫 Animated，使用 Animated 库，开发人员可以创造出流畅、强大并且易于构建和维护的动画。

在动画 API 中，系统为开发者提供了常用的动画效果组件包括：Animated.View、Animated.Text 和 Animated.Image 等，这些组件都是默认支持动画的。这些组件通过把动画数值绑定到组件的属性上，然后在逐帧执行原生更新，避免了每次渲染和同步过程的开销。

动画具有很强的可配置性，React Native 提供了多种动画属性，系统预定义了过渡函数、延迟、时间、衰减比例和刚度等动画效果。当然，还可以自定义动画，用以特殊的场合。原则上，一个 Animated.Value 可以驱动任意数量的属性，并且每个属性可以配置一个不同的插值函数。例如，我们希望把 Animated.Value 从 0 变化到 1，把组件的位置从 150px 移动到 0px，把不透明度从 0 变到 1，这时候可以通过修改 style 的属性来实现。

```
style={{
  opacity: this.state.fadeAnim, // Binds directly
  transform: [{
    translateY: this.state.fadeAnim.interpolate({
      inputRange: [0, 1],
      outputRange: [150, 0] // 0 : 150, 0.5 : 75, 1 : 0
    })
  }],
}}>
```

通过一些辅助函数，例如 `sequence` 或者 `parallel`（分别用于顺序执行多个动画和同时执行多个动画），还可以实现复杂的组合动画。在 Animated 中，通过给 `toValue` 设置为另一个

Animated.Value 来产生一个动画序列。

在 React Native 系统提供的动画方案中, Animated 模块被设计为可完全序列化的方式, 这样动画可以脱离 JavaScript 事件循环, 以一种高性能的方式运行。

除了上面介绍的函数之外, Animated 还提供了一些其他有用的辅助函数。例如, Animated.ValueXY 可实现 2D 动画, setOffset 和 getLayout 可辅助实现一些常见的页面交互等。读者可以通过官方提供的例子 AnimationExample.js 来学习相关知识。

Animated属性

Animated 主要提供了如下属性。

- Value

一个数值类型的类, 用于驱动动画。使用 new Animated.Value(0) 来初始化。

- ValueXY

一个 2D 值类型的类, 用来驱动 2D 动画。例如拖动、平移操作。

Animated方法

- static decay(value, config)

将一个值根据阻尼系数逐步衰减到 0。

- static timing(value, config)

根据时间函数来处理常见的动画曲线。例如, 线性、加速、减速等过渡曲线, 该方法还支持自定义时间函数。

- static spring(value, config)

产生一个基于rebound和origami实现的spring动画。它会在toValue值更新的同时跟踪当前的速度状态, 以确保动画连贯。

- static add(a, b)

将两个 Animated.value 相加, 创建一个新的动画值。

- static multiply(a, b)

将两个 Animated.value 相乘, 创建一个新的动画值。

- static modulo(a, modulus)

将 a 对 modulus (类似于 % 操作者), 创建一个新的动画值。

- static delay(time)

在指定的延迟之后启动动画。

- static sequence(animations)

顺序启动一组动画, 如果其中一个动画中途停止, 则整个动画组停止。

- static parallel(animations, config?)

同时启动一组动画, 如果一个动画中途停止, 则全都停止。可以通过设置stopTogether来重写这一特性。

- static stagger(time, animations)

以指定的延迟时间来启动组动画。

- `static event(argMAPPING, config?)`
接受一个映射的数组，用于手动控制动画的状态。
- `static createAnimatedComponent(Component)`
创建一个支持动画的 `Component`。

AnimatedValue

对于 `AnimatedValue` 而言，一个 `AnimatedValue` 一次可以驱动多个动画属性的类，但是，一个 `AnimatedValue` 一次却只能由一个机制驱动。例如，一个 `Value` 可以同时作用于动画的透明度和位置，但是一个 `Value` 一次却只能采用一种函数。`AnimatedValue` 提供的常用方法如下。

- `constructor(value)`
创建构造器。
- `setValue(value)`
给动画设置值，会导致动画终止。
- `setOffset(offset)`
给动画设置偏移量。
- `flattenOffset()`
将偏移量设为 0。
- `addListener(callback)`, `removeListener(id)`, `removeAllListeners()`
增加、删除动画监听。
- `stopAnimation(callback?)`
终止动画，并在动画结束后执行回调。
- `interpolate(config)`
动画插值器，可在动画更新前使用插值函数对当前值进行变换。

AnimatedValueXY

一个 2D 动画函数用来驱动 2D 动画。它提供的 API 和普通的 `Animated.Value` 所提供的几乎一样，只不过 `AnimatedValueXY` 提供的是一个动画的复合体（实际上包含两个普通的 `Animated.Value`）。除了和 `AnimatedValue` 有一些共用的方法之外，`AnimatedValueXY` 还提供了以下两个特有的方法。

- `getLayout()`
将一个 `{x, y}` 组合转换为 `{left, top}` 用于样式。例如 `style={this.state.anim.getLayout()}`。
- `getTranslateTransform()`
将一个 `{x, y}` 组合转换为一个可用的位移变换动画。例如 `style={{transform: this.state.anim.getTranslateTransform()}}`。

Animated使用

在正常情况下，创建一个动画需要如下步骤。

❶ 创建 `Animated.Value`，设置初始值，比如一个视图的 `opacity` 属性，最开始就设置为 `Animated.Value(0)`。

❷ 将 `Animated.Value` 绑定到组件的 `Style` 属性上。

❸ 使用 `Animated.timing` 创建自动的动画，或者通过 `Animated.event` 把它关联到一个手势上，如拖动或者滑动操作。

❹ 调用 `Animated.timing.start()` 来启动动画。

在创建动画方面，除了经常使用到的 `Animated.timing()` 和 `Animated.spring()`，还有 3 个比较独立的调用动画的方式。

- `Animated.parallel()`：同时开始一个动画数组里的全部动画。默认情况下，如果有任何一个动画停止了，其余的也会停止。
- `Animated.sequence()`：顺序执行一个动画数组里的动画。如果当前的动画被中止，后面的动画则不会继续执行。
- `Animated.stagger()`：同时执行动画数组里面的内容，可能会出现动画的重叠效果，可以设置动画的延迟来启动动画。

Animated示例

一款优秀的 APP，动画是必不可少的，包括简单的过渡、缩放、渐变动画，以及复杂的组动画等。复杂的组动画往往由简单动画构成。

Animated.timing()

使用 `Animated.timing` 可以创建旋转动画（见图 5-5）或者 Alpha 动画等效果。其使用方法如下：

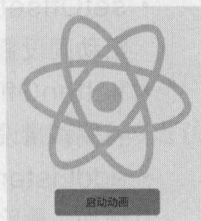


图5-5 旋转动画

```
Animated.timing(
  someValue,
  {
    toValue: number,
    duration: number,
    easing: easingFunction,
    delay: number
  })
```

在上面的例子中，`Easing` 是 React Native 创建动画的载体，它允许开发者使用已经定义好的各种缓冲函数。常见的缓冲函数有：`linear`、`ease`、`quad`、`cubic`、`sin`、`elastic`、`bounce`、`back`、`bezier`、`in`、`out` 和 `inout`。

作为一种经典的动画，旋转动画在很多项目中都有涉及，要让图片实现连续旋转，可以定义一个方法，在动画正常运行完成之后再次调用该方法来实现连续动画效果。核心代码如下：

```
componentDidMount () {
  this.spin()
}
// Animated 结束之后再次调用 spin()
spin () {
```

```

this.spinValue.setValue(0)
Animated.timing(
  this.spinValue,
  {
    toValue: 1,
    duration: 4000,
    easing: Easing.linear
  }
).start(() => this.spin())
}

```

使用 Animated.timing 创建旋转动画的完整代码如下:

```

class AnimationRotateScene extends Component {

  constructor(props) {
    super(props);
    this.spinValue = new Animated.Value(0)
  }

  componentDidMount () {
    this.spin()
  }

  spin () {
    this.spinValue.setValue(0)
    Animated.timing(
      this.spinValue,
      {
        toValue: 1,
        duration: 4000,
        easing: Easing.linear
      }
    ).start(() => this.spin())
  }

  render() {
    const
    spin = this.spinValue.interpolate({
      inputRange: [0, 1],
      outputRange: ['0deg', '360deg']
    })

    return (
      <View style={styles.container}>

        <Animated.Image
          style={{
            width: 227,
            height: 200,
            transform: [{rotate: spin}] }}

```

```

        source={{uri: 'https://s3.amazonaws.com/media-p.slid.es/
uploads/alexanderfarennikov/images/1198519/reactjs.png'}}
      />
      <TouchableOpacity onPress={() => this.spin()} style={styles.button}>
        <Text> 启动动画 </Text>
      </TouchableOpacity>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20,
    justifyContent: 'center',
    alignItems: 'center',
  },
  button: {
    marginTop: 20,
    backgroundColor: '#808080',
    height: 35,
    width: 140,
    borderRadius: 5,
    justifyContent: 'center',
    alignItems: 'center',
  },
});

export default AnimationRotateScene;

```

Animated.spring()

使用 Animated.spring 可以创建诸如缩放动画等动画效果。其核心 API 如下：

```

Animated.spring(
  someValue,
  {
    toValue: number,
    friction: number
  })

```

在具体使用过程中，通常会将会 Animated.spring 封装起来，然后调用 spring() 来启动动画。

```

spring () {
  this.springValue.setValue(0.3)
  Animated.spring(
    this.springValue,
    {
      toValue: 1,
      friction: 1
    }
  ).start()
}

```

在使用 `Animated.spring` 方法的过程中，有两个参数需要注意：一个是要变化的动画属性值和一个是可配置对象。可配置属性包括：`toValue` (number)、`overshootClamping` (boolean)、`restDisplacement Threshold` (number)、`restSpeed Threshold`(number)、`velocity` (number)、`bounciness` (number)、`speed` (number)、`tension`(number) 和 `friction` (number)，除了 `toValue` 参数的值是必需的之外，其他值都是可选的。

Animated.parallel()

使用 `Animated.parallel` 可以创建一个组动画，具体使用的时候，只需要将动画数组放到 `Animated.parallel` 里即可。其核心 API 如下：

```
Animated.parallel([
  Animated.spring(
    animatedValue,
    {
      //config options
    }
  ),
  Animated.timing(
    animatedValue2,
    {
      //config options
    }
  )
])
```

例如，下面是使用 `Animated.parallel` 实现组动画的例子。

```
Animated.parallel([
  componentDidMount () {
    this.animate()
  }
  animate () {
    this.animatedValue1.setValue(0)
    this.animatedValue2.setValue(0)
    this.animatedValue3.setValue(0)
    const createAnimation = function (value, duration, easing, delay = 0) {
      return Animated.timing(
        value,
        {
          toValue: 1,
          duration,
          easing,
          delay
        }
      )
    }
    Animated.parallel([
      createAnimation(this.animatedValue1, 2000, Easing.ease),
      createAnimation(this.animatedValue2, 1000, Easing.ease, 1000),

```



```

        createAnimation(this.animatedValue3, 1000, Easing.ease, 2000)
    }).start()
}

```

在本示例代码中，创建了一个 `createAnimation` 方法，该方法返回一个新的动画，该方法接受 4 个参数：`value`、`duration`、`easing` 和 `delay`（默认值 0）。然后，使用 `Animated.parallel()` 方法将创建的动画组合起来即可实现组动画效果。如果需要对动画做一些数字计算（如变速操作），还需要为动画设置插值。

```

render () {
    const scaleText = this.animatedValue1.interpolate({
        inputRange: [0, 1],
        outputRange: [0.5, 2]
    })
    const spinText = this.animatedValue2.interpolate({
        inputRange: [0, 1],
        outputRange: ['0deg', '720deg']
    })
    const introButton = this.animatedValue3.interpolate({
        inputRange: [0, 1],
        outputRange: [-100, 400]
    })
    ...
}

```

然后，调用 `Animated.Views` 即可完成界面的绘制。在本例中，借助 `Animated.parallel()` 函数，同时实现渐变、旋转、缩放效果。完整代码如下：

```

class AnimationGroupScene extends Component {
    constructor() {
        super()
        this.animatedValue1 = new Animated.Value(0)
        this.animatedValue2 = new Animated.Value(0)
        this.animatedValue3 = new Animated.Value(0)
    }

    componentDidMount() {
        this.animate()
    }

    animate() {
        this.animatedValue1.setValue(0)
        this.animatedValue2.setValue(0)
        this.animatedValue3.setValue(0)
        const createAnimation = function (value, duration, easing, delay = 0) {
            return Animated.timing(
                value,
                {
                    duration,
                    easing,
                    delay
                }
            )
        }
    }
}

```

```

        toValue: 1,
        duration,
        easing,
        delay
      }
    )
  }
  Animated.parallel([
    createAnimation(this.animatedValue1, 2000, Easing.ease),
    createAnimation(this.animatedValue2, 1000, Easing.ease, 1000),
    createAnimation(this.animatedValue3, 1000, Easing.ease, 2000)
  ]).start()
}

render() {
  const scaleText = this.animatedValue1.interpolate({
    inputRange: [0, 1],
    outputRange: [0.5, 2]
  })
  const spinText = this.animatedValue2.interpolate({
    inputRange: [0, 1],
    outputRange: ['0deg', '720deg']
  })
  const introButton = this.animatedValue3.interpolate({
    inputRange: [0, 1],
    outputRange: [-100, 400]
  })

  return (
    <View style={styles.container}>
      <Animated.View
        style={{transform: [{scale: scaleText}]}}>
        <Text>Welcome</Text>
      </Animated.View>
      <Animated.View
        style={{marginTop: 20, transform: [{rotate: spinText}]}}>
        <Text
          style={{fontSize: 20}}>
          to the App!
        </Text>
      </Animated.View>
      <Animated.View
        style={{top: introButton, position: 'absolute'}}>
        <TouchableHighlight
          onPress={this.animate.bind(this)}

```

```

        style={styles.button}>
        <Text> 启动组动画 </Text>
      </TouchableHighlight>
    </Animated.View>
  </View>
);
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20,
    justifyContent: 'center',
    alignItems: 'center',
  },
  button: {
    marginTop: 20,
    backgroundColor: '#808080',
    height: 35,
    width: 140,
    borderRadius: 5,
    justifyContent: 'center',
    alignItems: 'center',
  },
});
export default AnimationGroupScene;

```

除了上面介绍的动画方案之外，React Native 还提供 `Animated.sequence` 顺序组动画、`Animated.Stagger` 重叠动画等，借助这些动画方案，开发者可以实现复杂多变的动画效果。

5.7 Geolocation

作为移动开发中最基础的服务之一，定位被广泛应用在各种移动应用中。在原生应用的开发过程中，要实现定位和地图相关的功能就需要集成第三方库，如百度、高德、腾讯地图等。考虑到地图服务的重要性，Facebook 在设计 React Native 之初便想到了这些问题，索性在将这一功能直接集成了进去，因此在具体使用的时候，无需引入第三方库。

Geolocation 提供基本的定位信息和经纬度信息，而更加复杂的地图展示等则需使用 `MapView` 组件。使用 Geolocation 定位需要在应用中申请相关权限，对于 iOS，需要在 `Info.plist` 中增加 `NSLocationWhenInUseUsageDescription` 来开启定位权限；对于 Android，则需要在 `AndroidManifest.xml` 配置文件中增加定位权限 `ACCESS_FINE_LOCATION`。

Geolocation方法

Geolocation 提供的主要方法如下：

- `getCurrentPosition(geo_success, geo_error?, geo_options?)`
获取最新位置信息。成功后会调用 `geo_success` 回调函数，参数中包含最新的位置信息。支持的选项有：`timeout (ms)`、`enableHighAccuracy (bool)`、`maximumAge`

(ms)。选项的含义如下。

- enableHighAccuracy: 指示浏览器获取高精度的位置，默认为false。开启后，浏览器可能花费更长的时间获取更精确的位置数据。
- timeout: 指定获取地理位置的超时时间，默认不限时。
- maximumAge: 最长有效期。此参数用来指定多久再次获取位置。

请求成功后，函数返回如下属性。

- 经度：coords.longitude
- 纬度：coords.latitude
- 准确度：coords.accuracy
- 海拔：coords.altitude
- 海拔准确度：coords.altitudeAccuracy
- 行进方向：coords.heading
- 地面速度：coords.speed
- 时间戳：new Date(position.timestamp)

请求失败的原因有如下4种情况。

- 为用户拒绝定位请问。
- 暂时获取不到位置信息。
- 为请求超时。
- 未知错误。
- watchPosition(success, error?, options?)
位置变化监听，每当位置变化之后都调用success回调。
- clearWatch(watchID)
清除位置监听。

Geolocation示例

使用 Geolocation 获取的信息主要包括经纬度、速度、风向等，然后以 json 格式返回。需要注意的是，navigator.geolocation 中的 navigator 指的并不是导航器而是 Geolocation 的一个类。例如，下面使用 Geolocation 获取位置信息的例子，运行结果如图 5-6 所示。

```
getPosition() {
    navigator.geolocation.getCurrentPosition(
        (position) => {
            var initialPosition = JSON.stringify(position);
            console.log('initialPosition:' + initialPosition);
            this.setState({initialPosition});
            alert(initialPosition);
        },
        (error) => alert(JSON.stringify(error)),
    );
}
```

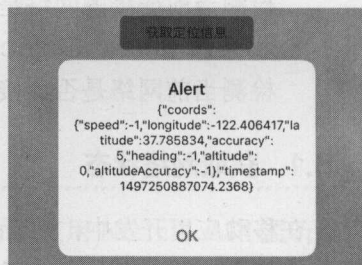


图5-6 Geolocation获取位置信息


```

    {enableHighAccuracy: true, timeout: 20000, maximumAge: 1000}
  );
  watchID = navigator.geolocation.watchPosition((position) => {
    var lastPosition = JSON.stringify(position);
    this.setState({lastPosition});
  });
}

```

5.8 NetInfo

在 React Native 中，使用系统提供的 `NetInfo` 接口可以获取手机当前的各个网络状态，从而针对不同的网络环境执行不同的页面处理。React Native 在初始化项目的时候，会默认为项目在项目目录下安装 `node-fetch` 依赖包，该依赖包是获取网络状态的重要工具。

属性和方法

- `addEventListener(eventName:ChangeEventName,handler:Function)`
用于设置网络变化事件监听器，需要传入回调的处理方法。
- `removeEventListener(eventName:ChangeEventName,handler:Function)`
用于移除网络事件变化监听器。
- `fetch()`
检测当前网络连接状态以及请求网络操作。
- `isConnectionExpensive(callback:(metered:?boolean,error?:string)=>void)`
检测当前连接的网路是否需要计费。
- `isConnected :ObjectExpression`
检测当前网络是否连接。

5.8.1 获取网络状态

在移动应用开发中，判断网络状态是开发中一个重要的环境，根据不同的网络环境，适配不同的显示结果，使用 `Netinfo.fetch` 函数正好可以满足这一需求。使用 `Netinfo` 获取当前网络状态的代码如下：

```

NetInfo.fetch().done((status)=> {
  console.log('Status: '+status);
});

```

Android获取网络状态

Android 手机获取网络状态，首先需要在应用的 `AndroidManifest.xml` 文件中添加网络状态权限。

```

<uses-permission android:name="android.permission.INTERNET" />
// 网络状态获取权限
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

```

添加完权限之后，直接重载代码是无效的，因为修改了 Android 原生的配置文件，所以需要重新执行 `react-native run-android` 安装命令。

Android 的网络状态比较多，主要有如下一些状态重新安装。

- NONE：设备处于离线状态。
- BLUETOOTH：蓝牙数据连接。
- DUMMY：模拟数据连接。
- ETHERNET：以太网数据连接。
- MOBILE：移动网络数据连接。
- MOBILE_DUN：拨号移动网络数据连接。
- MOBILE_HIPRI：高优先级移动网络数据连接。
- MOBILE_MMS：彩信移动网络数据连接。
- MOBILE_SUPL：安全用户面定位（SUPL）数据连接。
- VPN：虚拟网络连接（需要Android5.0以上）。
- WIFI：WIFI数据连接。
- WIMAX：WiMAX数据连接。
- UNKNOWN：未知数据连接。

iOS获取网络状态

iPhone 手机上获取网络状态，主要有如下几种状态。

- none：设备处于离线状态。
- wifi：设备处于联网状态且通过Wi-Fi链接，或者是一个iOS的模拟器。
- cell：设备通过Edge、3G、WiMax或是LTE网络联网。
- unknown：发生错误，网络状况不可知。

5.8.2 网络状态监听

NetInfo API 除了提供获取网络状态的功能外，开发者还可以使用 NetInfo API 监听网络状态改变事件。当网络状态发生改变时，应用程序会马上收到通知。

```
componentWillMount() {
  NetInfo.fetch().done((status)=> {
    console.log('网络状态 Status:' + status);
  });
  // 监听网络状态改变
  NetInfo.addEventListener('change', this.handleConnectivityChange);
}

componentWillUnmount() {
  console.log("componentWillUnmount");
  NetInfo.removeEventListener('change', this.handleConnectivityChange);
}
```

```

handleConnectivityChange(status) {
  console.log('status change:' + status);
  // 监听第一次改变后，可以取消监听
  NetInfo.removeEventListener('change', this.handleConnectivityChange);
}

```

5.8.3 判断网络是否连接

通过 NetInfo 提供的 isConnected 函数可以判断当前手机是否有网络连接。

```

NetInfo.isConnected.fetch().done((isConnected) => {
  console.log('First, is ' + (isConnected ? 'online' : 'offline'));
});

```

对于 Android 平台来说，NetInfo API 还提供额外的 isConnectionExpensive 函数来对网络连接是否收费进行判断。如果当前连接的是移动数据网络，那么可能会判定其为计费的数据连接。

```

NetInfo.isConnectionExpensive((isConnectionExpensive) => {
  console.log('Connection is ' + (isConnectionExpensive ? 'Expensive' : 'Not Expensive'));
});

```

5.9 小结

在本章中，主要介绍了 React Native 提供的常用 API，只有配合使用组件和 API，才能开发高质量的界面。本章在讲解基本 API 的基础之上，适当地增加了对 React Native 系统运作原理的讲解，这也是为后面自定义组件提供理论支撑。只有在应用和原理高度结合的情况下，开发者才能开发出高质量的移动应用。

组件封装

在原生 APP 开发过程中，系统提供给组件有时候并不能满足开发需求，这时我们首先想到的就是自定义组件，通过将自己编写的组件发布到第三方仓库还可以供其他有类似需求的开发者使用。本章将从常见的第三方库着手，讲解在 React Native 开发中常见的第三方库，进而学习封装组件。而在封装组件的过程中，对于组件的生命周期是必须掌握的。

6.1 组件的生命周期

所谓生命周期，就是一个对象从生成到消亡所经历的状态过程。理解组件的生命周期，是组件类应用开发的重要内容。在 React Native 中，组件都是有生命周期的，开发者可以根据组件所处的生命周期进行合理的需求开发，如图 6-1 所示，展示了组件生命周期的整个流程。组件的生命周期主要由 3 个部分组成：挂载、更新和移除。

- 挂载：组件被插入到 DOM 中。
- 更新：组件被重新渲染，检查 DOM 是否应该刷新。
- 移除：组件从 DOM 中移除。

挂载（初始化）

挂载又称为初始化，从调用 `getDefaultProps()` 开始到 `componentDidMount()` 结束。挂载阶段涉及的常见方法如下。

- `getInitialState()`

在组件被挂载前调用，状态化组件需实现此方法，并返回初始的状态。

- `componentWillMount ()`

在挂载发生之前立即被调用。

- `componentDidMount ()`

在挂载结束之后马上被调用，需要 DOM 节点的初始化操作调用此方法。

更新（运行）

当组件初始化完成之后，程序就开始正常运行起来，此时程序处于运行状态，在这个阶段涉及的常见方法如下。

- `componentWillReceiveProps(object nextProps)`

当一个挂载的组件接收到新的props的时候被调用。该方法应该用于比较this.props和nextProps，然后使用this.setState()来改变状态。

- `shouldComponentUpdate(object nextProps,object nextState)`

当组件需要更新DOM时被调用。通过this.state和nextState的比较来确定是否需要更新DOM，如果不需要React更新DOM，则返回false。

- `componentWillUpdate(object nextProps,object nextState)`

在更新发生之前调用。

- `componentDidUpdate(object prevProps,object prevState)`

在更新发生之后调用。

注意：此阶段最重要的一个概念就是所谓的状态机机制。从初始状态到状态的更新，从而触发界面的重新绘制。

移除（销毁）

移除阶段又称为销毁阶段，碰到以下情况会触发组件的移除操作：系统遇到错误崩溃、用户主动退出、系统内存不足等。移除阶段涉及的常见方法如下。

- `componentWillUnmount()`

在组件移除和销毁之前被调用。如果需要清理某些数据，可以调用此方法。

- `getDOMNode()`

DOM节点可以在任何挂载的组件上调用，用于获取一个指向它的DOM节点的引用。

- `forceUpdate()`

在一些嵌套很深的组件的状态已经改变的时候，可以通过此方法调用，而不是调用this.setState()。

React Native 组件生命周期常用函数调用次数如表 6-1 所示。

表 6-1 组件生命周期函数调用次数统计

方法及属性	调用次数	能否设置状态机
<code>getDefaultProps</code>	1（全局调用一次）	否
<code>getInitialState</code>	1	否

续表

方法及属性	调用次数	能否设置状态机
componentWillMount	1	是
render	≥ 1	否
componentDidMount	1	是
componentWillReceiveProps	≥ 0	是
shouldComponentUpdate	≥ 0	否
componentWillUpdate	≥ 0	否
componentDidUpdate	≥ 0	否
componentWillUnmount	1	否
getDefaultProps	1 (全局调用一次)	否

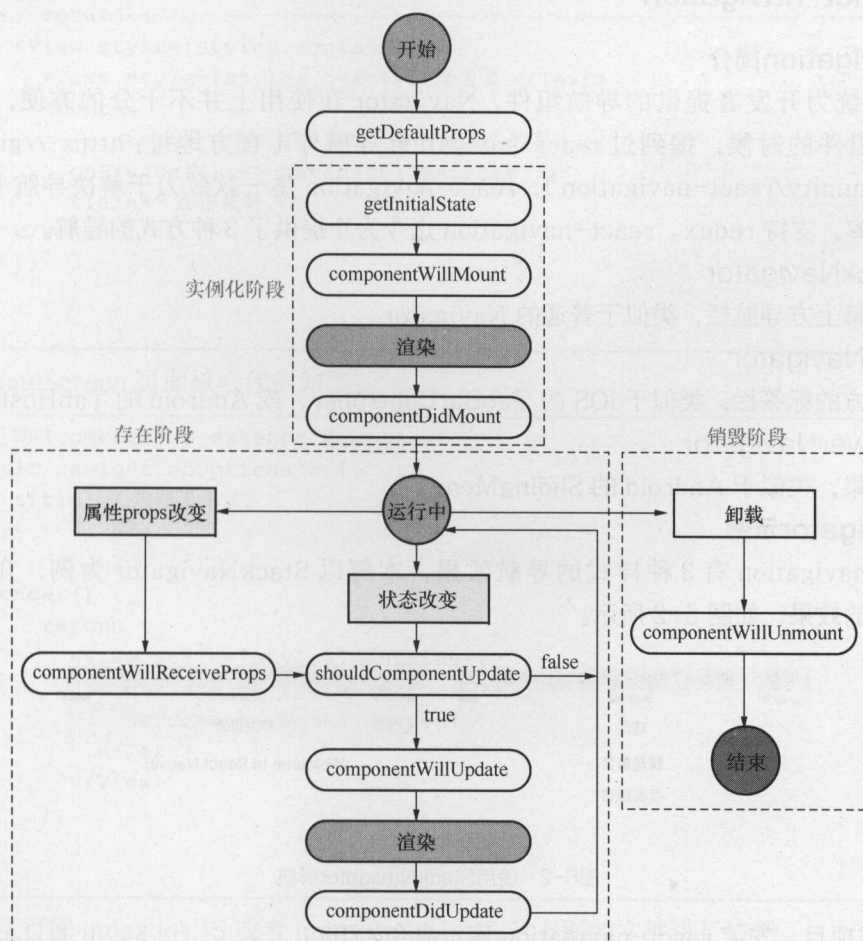


图6-1 组件生命周期流程示意图

在理解组件的生命周期的时候，应该注意以下几点。

- 组件创建后，如果不被调用，将默认调用`getDefaultProps()`方法。
- `componentWillMount()`和`componentWillUpdate()`是`render`调用之前的最后一个方法，可以用来处理`this.state`赋值操作以及业务逻辑操作。
- 因为`this.state`状态的改变，会重新渲染组件，所以不要在`render()`里做对`this.state`值有改变的操作，否则可能造成循环渲染，导致程序崩溃。

6.2 第三方库

在 React Native 项目开发过程中，常常需要借鉴一些开源的第三方库，这样，不仅可以加快开发速度，还可以提高代码的质量、减少代码的逻辑复杂度。

6.2.1 react-navigation

react-navigation简介

作为系统为开发者提供的导航组件，Navigator 在使用上并不十分的方便，前面在讲 Navigator 组件的时候，提到过 react-navigation 导航库（官方地址：<https://github.com/react-community/react-navigation>），react-navigation 是一款致力于解决导航卡顿、数据传递的导航库，支持 redux。react-navigation 迄今为止提供了 3 种方式的导航。

- StackNavigator

提供屏幕上方导航栏，类似于普通的 Navigator。

- TabNavigator

屏幕下方的标签栏，类似于 iOS 的 `TabBarController`，或 Android 的 `TabHost`。

- DrawerNavigator

抽屉效果，类似于 Android 的 `SlidingMenu`。

StackNavigator示例

react-navigation 有 3 种样式的导航效果，本篇以 StackNavigator 为例，介绍 react-navigation 的效果，如图 6-2 所示。

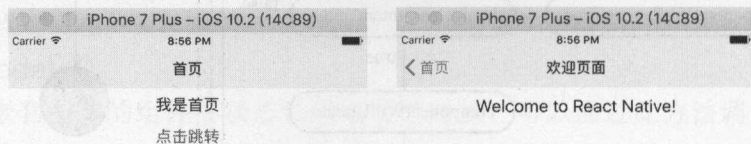


图6-2 使用StackNavigator导航

- ① 新建项目，安装 react-navigation 库，命令如下：

```
npm install --save react-navigation
```

- ② 新建两个页面，HomeScreen 作为当前页面，WelcomeScreen 作为跳转后页面。跳转

页面的核心用法如下:

StackNavigator(RouteConfigs, StackNavigatorConfig)

HomeScreen.js 页面核心代码如下:

```
import { StackNavigator } from 'react-navigation';
// 省略导包等
class HomeScreen extends Component {
  static navigationOptions = {
    title: '首页',
  };

  render() {
    const {navigate} = this.props.navigation;
    return (
      <View style={styles.container}>
        <Text style={styles.text}>我是首页</Text>
        <Button
          // navigate 里的字段对应在主页面定义的别名
          onPress={() => navigate('Welcome')}
          title="点击跳转"/>
        </View>
      );
    }
  }
}
```

WelcomeScreen 页面核心代码如下:

```
class WelcomeScreen extends Component {
  static navigationOptions = {
    title: '欢迎页面',
  };

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
      </View>
    );
  }
}
```

③ 在主页面 index.ios.JS 或者 index.android.JS 对上面的文件进行注册。

```
const SimpleAPP = StackNavigator({
  Home: { screen: HomeScreen },
  Welcome: { screen: WelcomeScreen },
```

```
});  
APPRegistry.registerComponent('Demo', () => SimpleAPP);
```

react-navigation其他属性

除了上面使用的几个属性和方法之外，StackNavigator 还有如下一些常见的属性和方法。

- header
用于标题相关属性的设置，如表6-2所示。

表 6-2 react-navigation 标题属性

方法及属性名	说明
visible	导航栏是否显示
title	导航栏的标题，字符串或者组件
backTitle	左上角返回键文字，默认是上一页标题
right	导航栏右按钮
left	导航栏左按钮
style	导航栏样式
titleStyle	导航栏标题样式
tintColor	导航栏颜色

- cardStack
通过配置card stack的gesturesEnabled属性，可以控制导航是否可以右滑返回。
- title
导航栏的标题。

StackNavigator参数属性

在使用 StackNavigator 进行导航的过程中，除了上面介绍的标题相关属性之外，还有一些比较实用的参数。

- initialRouteName
设置默认的页面组件，必须是在系统中已注册的页面组件。
- initialRouteParams
初始路由参数。
- navigationOptions
屏幕导航默认选项。
- paths
RouteConfigs 里面路径设置的映射。
- mode
页面切换模式，主要提供两种切换方式：card（左右切换）和 modal（上下切换）。
- headerMode
导航栏的显示模式，主要提供3种显示模式：float（默认）、screen（有渐变效果）和none（隐藏导航栏）。

- onTransitionStart
页面切换开始时的回调函数。
- onTransitionEnd
页面切换结束后的回调函数。

react-navigation 除了提供顶部导航 (StackNavigator) 之外, 还提供底部 Tab 导航 (TabNavigator) 和侧滑 (DrawerNavigator) 功能。关于后面两种导航方式, 读者可以自行研究。

6.2.2 react-native-tab-navigator

在移动设计交互中, 底部 Tab 导航 (又称标签导航), 作为一种经典的导航方式而被大多数移动应用采用。底部导航位于页面底部, 通常会包含 3 ~ 5 个子标签。在原生 APP 开发中, Android 的实现方式比较多, 诸如 TabHost+Activity、RadioButton+Fragment、Fragment TableHost+Fragment 等; 而 iOS 则可以通过 UITabBar+Controller 的方式来实现 Tab 导航。

在 React Native 中, 对于 iOS 平台系统提供了 TabBarIOS 和 TabBarIOS.Item 两个组件来实现 Tab 导航, 但是对于 Android, 系统则没有提供现成的组件。为了实现跨平台, 我们一般会选择第三方库, 而 react-native-tab-navigator 是一个不错的选择。

react-native-tab-navigator 属性及方法

对于 TabNavigator 来说, 主要有以下属性需要掌握。

- selected
当前选项卡是否被选中。
- title
选项卡标题。
- renderIcon
选项卡默认图标。
- renderSelectedIcon
选项卡选中图标。
- renderSelectedIcon
选项卡选中图标。
- badgeText
选项卡提示角标。
- onPress
选项卡点击响应事件。

react-native-tab-navigator 示例

在 React Native 中, 可以使用 react-native-tab-navigator 来开发 Tab 导航, 因为代码和业务的分离, 加之使用简单, 它得到很多开发者的喜爱。

如图 6-3 所示, 这是一个常见的页面切换方式, 要实现这种切换效果, 我们可以利用

react-native-tab-navigator 库来实现。

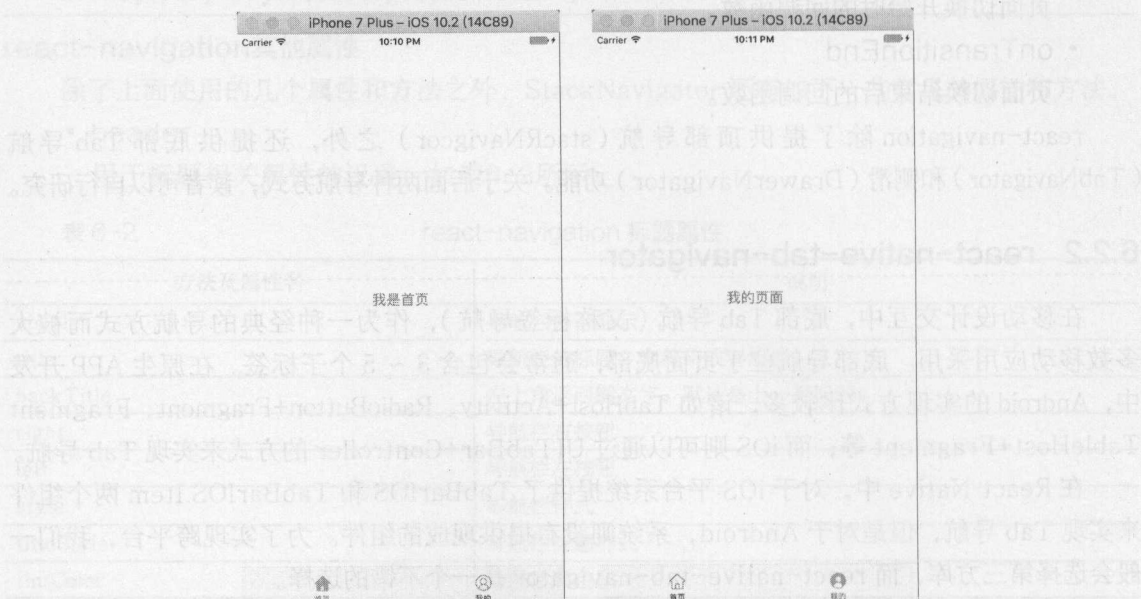


图6-3 Tab导航

- ① 打开终端，cd 到项目根路径。使用如下命令安装库：

```
npm install react-native-tab-navigator -save
```

- ② 在文件中导入 react-native-tab-navigator 包。

```
import TabNavigator from 'react-native-tab-navigator';
```

- ③ 编写代码：在 TabNavigator 中，每一个选项卡的组件由 TabNavigator.Item 构成，界面需要多少个 Tab，那么就有多少个 TabNavigator.Item 选项卡。如图 6-3 所示的界面由两个 TabNavigator.Item 组成。核心代码如下：

```
<TabNavigator.Item
  selected={this.state.selectedTab === 'Home'}
  title=" 首页 "
  titleStyle={styles.tabText}
  selectedTitleStyle={styles.selectedTabText}
  renderIcon={() => <Image style= {styles.icon}
source={require("./image/tabbar_homepage.png")} />}
  renderSelectedIcon={() => <Image style={styles.icon}
source={require("./image/tabbar_homepage_selected.png")} />}
  onPress={() => this.setState({ selectedTab: 'Home' })}}>
  <View style={styles.page}>
    <Text style={{fontSize:18,padding:15,color:
'red'}}> 我是首页 </Text>
```

```

    </View>
  </TabNavigator.Item>

```

在本例中，TabNavigator.Item 的子视图直接使用文字替代，而在商业项目中，TabNavigator.Item 则由具体的页面组成。

④ 设置默认选中项：

```

constructor(props) {
  super(props);
  this.state = {
    selectedTab: 'Home'
  }
}

```

使用 Tabnavigator 实现底部 Tab 导航的完整代码如下：

```

import React, { Component } from 'react';
import TabNavigator from 'react-native-tab-navigator';
import {
  APPRegistry,
  StyleSheet,
  Text,
  Image,
  View
} from 'react-native';

export default class TabNavigator extends Component {
  constructor(props) {
    super(props);
    this.state = {
      selectedTab: 'Home'
    }
  }

  render() {
    return (
      <View style={styles.container}>
        <TabNavigator>
          <TabNavigator.Item
            selected={this.state.selectedTab === 'Home'}
            title="首页"
            titleStyle={styles.tabText}
            selectedTitleStyle={styles.selectedTabText}
            renderIcon={() => <Image style={styles.icon}
source={require("../image/tabbar_homepage.png")} />}
            renderSelectedIcon={() => <Image style={styles.icon}
source={require("../image/tabbar_homepage_selected.png")} />}

```



```

        onPress={() => this.setState({ selectedTab: 'Home' })}}>
        <View style={styles.page}>
            <Text style={{fontSize:18,padding:15,color:
'red'}}>>我是首页 </Text>
        </View>
    </TabNavigator.Item>
    <TabNavigator.Item
        selected={this.state.selectedTab === 'Mine'}
        title="我的 "
        titleStyle={styles.tabText}
        selectedTitleStyle={styles.selectedTabText}
        renderIcon={() => <Image style={styles.icon} source=
{require("./image/tabbar_mine.png")} />}
        renderSelectedIcon={() => <Image style={styles.icon}
source={require("./image/tabbar_mine_selected.png")} />}
        onPress={() => this.setState({ selectedTab: 'Mine' })}}>
        <View style={styles.page}>
            <Text style={{fontSize:18,padding:15,color:
'red'}}>>我的页面 </Text>
        </View>
    </TabNavigator.Item>
</TabNavigator>
</View>
);
}
}

const styles = StyleSheet.create({
    container: {
        flex: 1
    },
    tabText: {
        fontSize: 10,
        color: 'black'
    },
    selectedTabText: {
        fontSize: 10,
        color: 'green'
    },
    icon: {
        width: 22,
        height: 22
    },
    page: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#FFFFFF'
    },
});

```

```
});
```

```
AppRegistry.registerComponent(TabNavigator, () => TabNavigator);
```

6.2.3 react-native-scrollable-tab-view

在 React Native 移动应用开发中，由于 Android 系统和 iOS 系统平台的差异，导致并不是所有的组件都能同时满足两个平台的需要，这时候就只能差异化对待。在页面移动切换方面，Android 和 iOS 就是存在差异的，对待这方面，iOS 平台用的是 TabBarIOS 组件，而 Android 平台则用的是 ViewPagerAndroid 组件，当然，另一种更通用的方式是使用第三方库，例如 react-native-scrollable-tab-view（官方开源地址：<https://github.com/skv-headless/react-native-scrollable-tab-view>）。使用 react-native-scrollable-tab-view 可以实现顶部和底部 Tab 切换。

其实，react-native-scrollable-tab-view 的底层实现上，就是通过获取不同的平台，然后去调用平台的组件来实现页面切换的。具体来说，在 Android 平台上调用 ViewPagerAndroid 组件，在 iOS 平台调用 TabBarIOS 组件。

react-native-scrollable-tab-view 属性及方法

react-native-scrollable-tab-view 常用的属性和方法如下。

- renderTabBar(Function:ReactComponent)

系统提供了两种默认的 TabBar 样式，分别是 DefaultTabBar 和 ScrollableTabBar。为了方便，本例中子视图使用文字替换。示例代码如下：

```
<ScrollableTabView
  renderTabBar={() => <DefaultTabBar/>}>
  <Text tabLabel='Tab1' />
  <Text tabLabel='Tab2' />
</ScrollableTabView>
```

TabBar 提供的主要样式如下。

- DefaultTabBar: Tab 会平分水平方向的空间。
- ScrollableTabBar: Tab 可以超过屏幕范围，可以滚动显示。
- tabBarPosition (String, 默认值为 top)

本属性用于控制 TabBar 显示的位置。

- top: 显示在屏幕顶部。
- bottom: 显示在屏幕底部。
- overlayTop: 显示在屏幕顶部，悬浮在内容视图之上（看颜色区分：视图有颜色，Tab 栏没有颜色）。
- overlayBottom: 位于屏幕底部，悬浮在内容视图之上（看颜色区分：视图有颜色，Tab 栏没有颜色）。
- onChangeTab()

在Tab切换之后会触发此方法，回调参数包含一个Object对象，我们可以从这个对象中获取。

- i: 被选中的Tab的下标（从0开始）。
- ref: 被选中的Tab对象。
- onScroll()

在视图滑动的时候触发此方法，返回一个Float类型的数字，其范围是[0, tab的数量-1]。

- locked (Bool, 默认为false)

本属性表示能否通过滑动来切换视图，默认为false表示可以滑动切换。如果为true，只能通过点击Tab来切换。

- initialPage(Integer)

初始化时被选中的Tab下标，默认值为0。

- page(Integer)

设置指定的Tab选中。

- tabBarUnderlineStyle(style)

设置DefaultTabBar和ScrollableTabBarTab选中时下方横线的颜色。

- tabBarBackgroundColor(String)

设置Tab的背景颜色。

- tabBarActiveTextColor(String)

设置选中Tab的文字颜色。

- tabBarInactiveTextColor(String)

设置未选中Tab的文字颜色。

- tabBarTextStyle(Object)

设置Tab文字的样式，比如文字的字体颜色等。

- contentProps(Object)

使用新的属性去覆盖ScrollView或ViewPagerAndroid默认的属性，系统不建议使用这个属性。

- scrollWithoutAnimation (Bool, 默认为false)

点击Tab切换时，是否显示动画效果。默认不使用动画。

react-native-scrollable-tab-view示例

react-native-scrollable-tab-view既可以实现顶部导航栏切换，也可以实现底部标签导航切换。例如，图6-4所示是新闻类应用常见的切换效果。

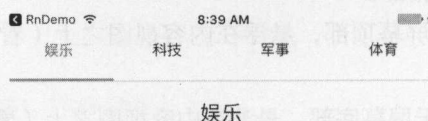


图6-4 顶部导航栏

❶ 首先，在项目目录下使用 npm 命令安装 react-native-scrollable-tab-view 库，安装命令如下：

```
npm install react-native-scrollable-tab-view --save
```

❷ 然后，在需要使用的页面文件中导入库。

```
import ScrollableTabView, {DefaultTabBar, ScrollableTabBar} from 'react-native-scrollable-tab-view';
```

❸ 编写切换逻辑代码：

```
<ScrollableTabView
  style={styles.container}
  renderTabBar={() => <DefaultTabBar />}
  tabBarUnderlineStyle={styles.lineStyle}
  tabBarActiveTextColor='#FF0000'>

  <Text style={styles.textStyle} tabLabel='娱乐'>娱乐</Text>
  <Text style={styles.textStyle} tabLabel='科技'>科技</Text>
  <Text style={styles.textStyle} tabLabel='军事'>军事</Text>
  <Text style={styles.textStyle} tabLabel='体育'>体育</Text>
</ScrollableTabView>
// 省略...

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20
  },
  lineStyle: {
    width: ScreenWidth/4,
    height: 2,
    backgroundColor: '#FF0000',
  },
  textStyle: {
    flex: 1,
    fontSize: 20,
    marginTop: 20,
    textAlign: 'center',
  },
});
```

在上面的示例代码中，为了简单，直接使用 Text 标签作为被包含的子视图，而在实际开发中，子视图的内容往往由更加复杂的页面元素构成。例如：

```
import HomeScreen from './widght/HomeScreen';
import MineScreen from './widght/MineScreen';
// 省略...

<ScrollableTabView
  style={styles.container}
  renderTabBar={() => <DefaultTabBar />}>
  // HomeScreen、MineScreen 是业务界面
```

```

<HomeScreen style={styles.textStyle} tabLabel=' 娱乐 '>娱乐</HomeScreen>
<MineScreen style={styles.textStyle} tabLabel=' 科技 '>科技</MineScreen>
</ScrollableTabView>

```

react-native-scrollable-tab-view 除了可以实现顶部导航栏效果外,还可以实现底部 Tab 导航切换,类似于 react-native-tab-navigator 实现的底部 Tab 切换效果,如图 6-5 所示。使用 react-native-scrollable-tab-view 作为底部导航栏时,需要对 tabBarPosition 进行设置(默认为 top)。由于 react-native-tab-navigator 默认采用文字加横线指示器的默认样式,如果应用在底部导航栏,还需要对 renderTabBar 函数进行设置。

使用 react-native-tab-navigator 实现底部导航栏切换的代码如下:

首页页面



图6-5 底部导航栏

```

import React, {Component} from 'react';
import ScrollableTabView from 'react-native-scrollable-tab-view';
import TabBottom from '../component/TabBottom';
import HomeScreen from './HomeScreen'; // 首页
import MineScreen from './MineScreen'; // 我的页面
const tabTitles = [' 首页 ', ' 我的 '];
// 默认图标
const tabIcon = [
  require('../images/tabbar_homepage.png'),
  require('../images/tabbar_mine.png'),
];
// 选中图标
const tabSelectedIcon = [
  require('../images/tabbar_homepage_selected.png'),
  require('../images/tabbar_mine_selected.png'),
];

export default class TabDefaultView extends Component {

  onChangeTabs = ({i}) => 'light-content';

  render() {
    return (
      <ScrollableTabView
        renderTabBar={() =>
          <TabBottom
            tabNames={tabTitles}
            tabIconNames={tabIcon}
            selectedTabIconNames={tabSelectedIcon}/>
          </TabBottom>
        }
        tabBarPosition='bottom'
        onChangeTab={this.onChangeTabs}>

```

```
<HomeScreen navigator={this.props.navigator}/>
<MineScreen navigator={this.props.navigator}/>
```

```
</ScrollableTabView>
```

```
); }
```

```
}
```

其中, TabBottom 为 TabBar 自定义的样式文件, 系统默认提供了两种样式: 分别是 DefaultTabBar 和 ScrollableTabBar, HomeScreen 和 MineScreen 为具体的展示页面。

6.2.4 react-native-image-picker

在移动应用开发中, 拍照和图片选取是常见的功能。虽然系统给开发者提供了 CameraRoll 组件, 但是由于其实现上比较复杂并且使用也不是很方便, 所以, 它并不被开发者所接受, 笔者推荐一款使用频率较高的第三方库, react-native-image-picker。

react-native-image-picker 示例

我们使用 react-native-image-picker 完成如下示例: 选择照片或拍照, 并将结果显示在界面上。效果如图 6-6 所示。

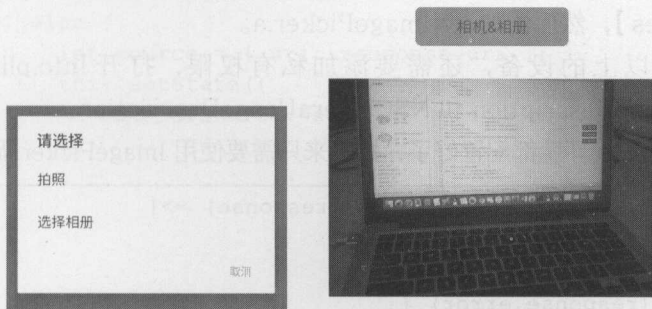


图6-6 Android选择图片并展示

首先, 使用命令导入 react-native-image-picker 库:

```
npm install react-native-image-picker@latest --save
```

由于 Android 和 iOS 平台对于包管理方式的差异, 所以在使用 ImagePicker 的时候, 需要分别过 Android 和 iOS 平台进行不同的依赖配置。

Android 环境配置

Android 平台集成 react-native-image-picker 的流程如下。

① 修改 Android 目录下的 setting.gradle 文件:

```
include ':react-native-image-picker'
project(':react-native-image-picker').projectDir = new File(rootProject.
projectDir, '../node_modules/react-native-image-picker/android')
```

- ② 在 Android/app 的 build.gradle 的 dependencies 节点下导入库文件:

```
compile project(':react-native-image-picker')
```

- ③ 在 MainApplication.java 类的 getPackages() 中添加如下语句 (第二步完成后, 系统默认会添加): new Image PickerPackage()。添加后代码如下:

```
@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
        new MainReactPackage(),
        new ImagePickerPackage()
    );
}
```

iOS环境配置

iOS 平台集成 react-native-image-picker 的方式和添加原生第三方库的方式一致, 大体需要如下步骤。

- ① 打开 Xcode, 右键单击项目, 选择【Add Files to xxx】, 然后依次选择【node_modules】→【react-native-image-picker】→【iOS】, 打开 ios 文件选中【RNImagePicker.xcodeproj】。
- ② 添加进来后, 选择 link 添加项目依赖, 选中项目, 选择【Build Phases】→【Link Binary With Libraries】, 然后添加 RNImagePicker.a。
- ③ 对于 iOS 10 以上的设备, 还需要添加私有权限, 打开 Info.plist 添加如下权限: NSPhotoLibrary UsageDescription 和 NSCameraUsageDescription。

至此, Android 和 iOS 环境都配置好了, 接下来只需要使用 ImagePicker 调用相关方法即可。

```
ImagePicker.showImagePicker(options, (response) =>{
    if (response.didCancel) {
        alert("用户点击了取消:");
    } else if (response.error) {
        alert("ImagePicker 发生错误: " + response.error);
    } else {
        let source = { uri: response.uri };
        this.setState({
            avatarSource: source
        });
    }
})
```

使用 imagepiczzer 完成拍照和选图的完整代码如下:

```
// 设置弹框界面
var options = {
    title: '请选择',
    cancelButtonTitle: '取消',
    takePhotoButtonTitle: '拍照',
    chooseFromLibraryButtonTitle: '选择相册',
    quality: 0.75,
```

```

allowsEditing:true,
noData:false,
storageOptions: {
  skipBackup: true,
  path:'images'
}
});

class ImagePickerView extends Component {

  constructor(props) {
    super(props);
    this.state = {
      avatarSource: null
    };
  }

  openMycamera = () =>{
    ImagePicker.showImagePicker(options, (response) =>{
      if (response.didCancel) {
        alert("用户点击了取消:");
      } else if (response.error) {
        alert("ImagePicker 发生错误:" + response.error);
      } else {
        let source = { uri: response.uri };
        this.setState({
          avatarSource: source
        });
      }
    })
  }

  render() {
    return (
      <View style={styles.container}>
        <TouchableOpacity style={styles.button} onPress={() => this.
openMycamera()} >
          <Text >相机 & 相册 </Text>
        </TouchableOpacity>
        <Image source={this.state.avatarSource} style={styles.
imageStyle} />
      </View>
    );
  }

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      marginTop: 20,
      alignItems: 'center',

```



```

    },
    button: {
      marginTop: 20,
      backgroundColor: '#808080',
      height: 35,
      width: 140,
      borderRadius: 5,
      justifyContent: 'center',
      alignItems: 'center',
    },
    imageStyle: {
      height: 180,
      width: 250,
      marginTop: 30,
      alignSelf: 'center',
    },
  },
});

export default ImagePickerView;

```

除了上面提到的一些属性之外，react-native-image-picker 还提供如下属性。

- customButtons
自定义弹窗的按钮。
- cameraType
指定拍照过程中使用前置还是后置摄像头。
- maxWidth/ maxHeight
设置图片的最大宽度/高度。
- quality
图片的质量（0-1）。
- storageOptions
设置图片存储位置，如果设置此属性，图像将保存在iOS的应用程序文档目录或是Android上应用程序的图片目录（而不是临时目录）。
- storageOptions.path
设置保存图片的路径。
- storageOptions.cameraRoll
如果设置此属性，裁剪后的图片将保存到系统默认路径。

上传服务器

在商业项目开发中，当用户选完图片之后，必然会涉及将图片同步到服务器实现持久化。本文提供一种比较简单的技术实现：使用 Promise 封装带有 FormData 对象的 fetch 请求，来完成将图片上传到图片服务器的功能。其核心代码如下：

```

let common_url = 'http://192.168.1.1:8080/'; // 服务器地址
let token = ''; // 用户登录后返回的 token

```

```
// 封装带请求参数的图片上传方法
function uploadImage(url,params){
  return new Promise(function (resolve, reject) {
    let formData = new FormData();
    for (var key in params){
      formData.append(key, params[key]);
    }
    let file = {uri: params.path, type: 'application/octet-stream', name:
'image.jpg'};
    formData.append("file", file);
    fetch(common_url + url, {
      method: 'POST',
      headers: {
        'Content-Type': 'multipart/form-data;charset=utf-8',
        "x-access-token": token,
      },
      body: formData,
    }).then((response) => response.json())
    .then((responseData) => {
      console.log('uploadImage', responseData);
      resolve(responseData);
    })
    .catch((err)=> {
      console.log('err', err);
      reject(err);
    });
  });
}
```

在具体的使用场景中，只需要按照和服务器约定的规范传入接口地址和参数即可。相关代码如下：

```
// 请求参数
let params = {
  userId:'abc12345',    // 用户 id
  path:'file:///storage/emulated/0/Pictures/image.jpg'  // 本地文件地址
}
uploadImage('app/uploadFile',params )
  .then( res=>{
    if(res.header.statusCode == 'success'){
      // 请求成功，进行相关逻辑处理
    }else{
      // 异常和失败处理
    }
  }).catch( err=>{
    // 请求失败，异常处理
  })
```

当然，在实际开发过程中，有时候还会涉及图片的裁剪操作，这里推荐一款比较实用的第

三方选图裁剪库: react-native-image-crop-picker。它的强大之处在于,它可以同时支持图片的单选和多选,使用上和 react-native-image-picker 大体相似,也需要分别配置 Android 平台和 iOS 平台依赖环境,然后定制开发。

6.2.5 Mobx

在使用 React 开发一些小应用的时候,数据、业务逻辑和视图的模块划分往往不是那么清楚,代码经常糅合在一起,这对于大型的应用程序来说往往是致命的。在中大型应用中,如果数据、业务逻辑和视图等模块得不到很好的层次划分,就会造成后期维护上的困难,甚至带来一系列的开发问题。

要实现模块开发和分层开发就会涉及组件的通信问题,于是,redux 应运而生。现在很多前端大型项目中都会看到 redux 的影子。不过,redux 在使用过程中比较复杂和繁琐,所以,推荐一款使用更简单、更灵活的状态管理框架 Mobx。

Mobx 作为一款状态管理工具,由 redux 作者亲荐。React 关注的是从状态 (state) 到视图 (view) 的问题,而 Mobx 关注的是状态仓库 (store) 到状态 (state) 的问题。

在介绍 Mobx 之前,需要重点理解几个核心概念。

状态 (State)

状态即数据,包括从服务端获取的数据,以及本地控制组件状态的数据。在与用户的交互过程中,状态会发生变化,当状态改变后,界面会重新绘制,从而让用户界面和数据保持一致。

派生 (Derivations)

由 state 演变而来,无需任何进一步交互的都统称为派生。派生存在的形式有很多,如下所示。

- 用户接口。
- 派生数据,例如列表的长度。
- 后端集成,例如发起网络请求。

在 Mobx 中,主要有两种类型的派生。

- 计算属性:基于state计算出的一些属性,例如,计算一个任务列表的长度。
- Reactions:当state改变时,Reactions会自动执行,例如,有时候需要在一个状态改变后才能执行某些操作。

Action

Action 是纯粹的 JavaScript 对象。Action 必须有一个 type 属性,type 能指示所执行的 action 的类型。对于一些小的应用,没有必要在分离文件中定义 action type 常量。

明确程序中定义的动作含义,可以让代码结构更加清晰。在严格模式下,修改 state 的函数最好包裹在 action 内,这样可以清楚地监听状态的变化,从而方便对状态的维护。

Mobx基本概念

Mobx 作为一款状态管理工具,目前只支持单向数据流格式。Mobx/MobxReact 中有 3 个核心概念:observable、observer、action。

当状态改变时,所有的派生都会自动更新,这意味着 Mobx 永远都不可能观察到中间值。

所有的派生默认都是同步更新的，这得益于 action 能够在修改状态后，安全可靠地观察计算属性。为了说明它们之间的关系，这里提供一个简化的关系图，如图 6-7 所示。

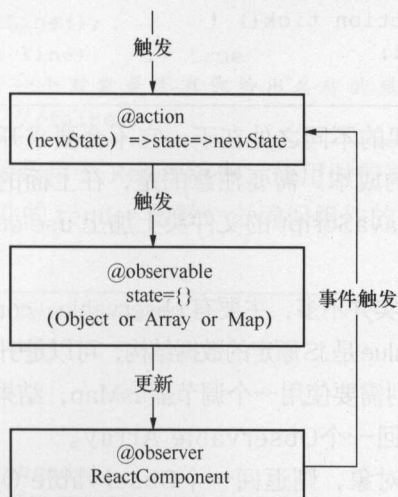


图6-7 Mobx简化关系图

Mobx 最核心的对象莫过于观察者和被观察者，对应于 Mobx 的 @observer 和 @observable。使用 @observable 将对象变为一个被观察者。例如：

```
import {observable} from 'mobx'
let appState = observable({
  timer: 0
})
```

现在，不需要让 appState 去观察什么，只需要创建视图（View），然后使用 @observer 标签将组件变为观察者即可，当 appState 的数据发生变化的时候，Mobx 会采用最优的方式去更新视图。例如下面是 mobx-react 的使用示例。

```
import {observer} from 'mobx-react';
@observer
class TimerView extends React.Component {
  render() {
    return (<button onClick={this.onReset.bind(this)}>
      Seconds passed: {this.props.appState.timer}
    </button>);
  }
  onReset () {
    this.props.appState.resetTimer();
  }
};
React.render(<TimerView appState={appState} />, document.body);
//resetTimer() 相关代码
```



```

appState.resetTimer = action(function reset() {
  appState.timer = 0;
});
setInterval(action(function tick() {
  appState.timer += 1;
}), 1000);

```

Mobx 和其他状态管理框架的不同之处在于，它不会要求开发者去手动维护程序开发中的相关状态，从而节约状态维护的成本。需要注意的是，在上面的例子中，使用 `@action` 标签改变数据的状态的时候，记得在 JavaScript 的文件头上加上 `use strict` 声明。

Mobx API

目前，Mobx 提供的 API 其实并不多，主要有 `observable`、`computed`、`reactions` 和 `actions`。

`observable(value)`: 其中 `value` 是 JS 原定的数据结构，可以是引用、对象、数组、ES6 的 `map` 等。

- 如果 `value` 是一个 `map`，则需要使用一个调节器 `asMap`，结果会返回一个 `Observable Map`。
- 如果是一个数组，则返回一个 `Observable Array`。
- 如果是一个没有属性的对象，则返回一个 `Observable Object`。
- 如果是一个有属性的对象、JavaScript 原有的数据结构、函数等，则返回一个 `Boxed Observable`。

以下使用 `Observable` 的一些例子：

```

const map = observable(asMap({ key: "value" }));
map.set("key", "new value");

```

```

const list = observable([1, 2, 4]);
list[2] = 3;

```

```

const person = observable({
  firstName: "Clive Staples",
  lastName: "Lewis"
});
person.firstName = "C.S.";

```

```

const temperature = observable(20);
temperature.set(25);

```

`@observable`：此标签用于检测需要变化的数据。

```

import {observable} from "mobx";
class OrderLine {
  @observable price:number = 0;
  @observable amount:number = 1;
  constructor(price) {
    this.price = price;
  }
  @computed get total() {

```

```

    return this.price * this.amount;
  }
}
const line = new OrderLine();
console.log("price" in line); // true
//hasOwnProperty: 判断一个对象是否有你给出名称的属性或对象。console.log(line.
hasOwnProperty("price")); //false

```

@observer: observer 函数用于 React 组件，使用时需要导入 mobx-react 依赖包。它通过 mobx.autorun 来包装组件的 render 函数，以确保组件的 render 函数在任何数据更改后都会强制重新渲染界面。

```

import {observer} from "mobx-react";
var timerData = observable({
  secondsPassed: 0
});
setInterval(() => {
  timerData.secondsPassed++;
}, 1000);
@observer class Timer extends React.Component {
  render() {
    return (<span>Seconds passed: { this.props.timerData.secondsPassed } </span> )
  }
};
React.render(<Timer timerData={timerData} />, document.body);

```

在使用 observer 函数时需要注意，如果还有其他的衍生物（decorators）或者高阶组件存在，请确保 observer 为最内层的修饰器否则它将无法正常工作。

Mobx 能做很多事情，但是它却不能把原始的数据变成观察者，（但它可以通过包裹这个值来返回一个 boxed observables 对象）。所以观察者观察的不是这个原始值，而是返回对象的属性值。

如果想把组件的内部状态变为可观察的，可以在组件内部使用 @observer 修饰器。

除此之外，mobx-react 提供的 Provider 组件还可以把传递下来的属性作用在 React 提供的上下文。通过连接这些 stores 和 observer，observer 将被作为组件的属性来使用。

```

const colors = observable({
  foreground: '#000',
  background: '#fff'
});
const App = () =>
  <Provider colors={colors}>
    <app stuff... />
  </Provider>;
const Button = observer(["colors"], ({ colors, label, onClick }) =>
  <button style={{
    color: colors.foreground,
    backgroundColor: colors.background

```

```

    })
    onClick={onClick}
  >{label}<button>
);
// later..
colors.foreground = 'blue';
// all buttons updated

```

@computed : computed 就像一个算术公式一样，从现有的状态或其他值去计算出需要值，而计算过程往往是耗时的。computed 采用了高度优化的算法，会尽可能帮你减少耗费。当参与计算的值没有发生改变，computed 是不会重新运行，并且不会触发界面的重绘。如果 computed 不再是观察者，那么系统会把它回收。Mobx 能自动进行垃圾回收。

```

class Foo {
  @observable length: 2,
  @computed get squared() {
    return this.length * this.length;
  }
  set squared(value) { //this is automatically an action, no annotation necessary
    this.length = Math.sqrt(value);
  }
}

```

autorun : autorun 被用在一些想要产生一个不用观察者参与的被动函数里。当 autorun 被调用的时候，一旦依赖项发生变化，autorun 提供的函数就会被执行。与之相反，computed 提供的函数只会是在有自己的观察员 (observers) 的时候才会评估是否需要重新执行。

所以，如果需要一个自动运行但不会产生任何新的值作为结果，那么请使用 autorun，其他情况请使用 computed。autorun 只用于达到某种结果，而不是强调计算的结果。

```

var numbers = observable([1,2,3]);
var sum = computed(() => numbers.reduce((a, b) => a + b, 0));
var disposer = autorun(() => console.log(sum.get()));
// 输出 '6'
numbers.push(4);
// 输出 '10'
action

```

任何应用程序都会涉及到对状态的操作，使用 action 标签可以改变任何事物的状态。在 Mobx 框架中，使用 @action 标签可以监控数据改变的自定义方法。在需要数据改变的地方执行此自定义方法，那么视图就会跟着自动变化。

细心的读者可能会问，React 官方不是说过不能直接修改 props 值嘛，但是为什么引入 Mobx 后 props 是可以直接修改的，那 Mobx 是如何实现的呢？首先看一下 @action 的用法。

```

// 定义一个 action
@action

```

```

createRandomContact() {
  this.pendingRequestCount++;
  superagent
    .get('https://randomuser.me/api/')
    .set('Accept', 'application/json')
    .end(action("createRandomContact-callback", (error, results) => {
      if (error) console.error(error)
      else {
        const data = JSON.parse(results.text).results[0];
        const contact = new Contact(this, data.dob, data.name,
data.login.username, data.picture)
        contact.addTag('random-user');
        this.contacts.push(contact);
        this.pendingRequestCount--;
      }
    }
  ))
}

```

对于 action 而言，仅仅可以作用于当前函数，而不能作用于当前函数调用的其他函数。这意味着在一些定时器任务中或者异步任务处理（如网络请求）的情况下，它们的回调函数无法将对状态作出改变。事实上，在 Mobx 框架中，这些回调函数都应该由 action 标签包裹，如例子中的 createRandomContact-callback。但是，如果你想通过 async/await 来改变状态的话，最好使用 runInAction，这样更简单。

```

@action
updateDocument = async () => {
  const data = await fetchDataFromUrl();
  /* required in strict mode to be allowed to update state: */
  runInAction("update state after fetching data", () => {
    this.data.replace(data);
    this.isSaving = true;
  })
}

```

目前看到的 action 都遵循在 JavaScript 中绑定的正常规则，但是在 Mobx 3 中引入了 action.bound 来自动绑定 action 到目标对象上。和 action 使用不一样的是，action.bound 并不需要名字参数。例如：

```

class Ticker {
  @observable this.tick = 0
  @action.bound
  increment() {
    this.tick++ // 'this' will always be correct
  }
}

const ticker = new Ticker()
setInterval(ticker.increment, 1000)

```

Mobx 作为一款既简单又可扩展的状态管理库，很好地实现了对状态（state）的监听。应用程

序的 state 是最基础的数据，它不应该包含冗余和派生的数据。action 是唯一能够改变 state 的方法。

派生值（computed values）通过纯函数从 state 中派生而来，当派生值依赖的状态发生变化时，Mobx 将会自动更新派生值。如果依赖的状态没有改变，Mobx 会做优化处理。

在前端开发中，React 和 Mobx 是非常强大的组合。React 提供了将应用状态映射为可渲染的组件树的机制，Mobx 提供存储和更新应用状态的机制，供 React 使用。React 通过使用虚拟 DOM，减少了对浏览器 DOM 的操作。Mobx 则通过使用响应式虚拟依赖状态图（reactive virtual dependency state graph），提供了应用程序状态与 React 组件同步的机制，减少了对页面的绘制。

Mobx与Redux对比

Mobx 的优势在于可变数据（Mutable Data）和可观察数据（Observable Data），而 Redux 的优势则在于不可变数据（Immutable data）。可变数据和不可变数据的区别在于，可变数据创建后可以修改，不可变数据创建后不可以修改。可变数据，可以直接对数据的状态进行修改，所以操作起来非常简单。不可变数据只要创建后便不可修改，不可变数据也并不一定会用到 Immutable 库。不可变数据的优势在于：可预测、可回溯。例如：

```
function foo(bar) {
  let data = { key: 'value' };
  bar(data);
  console.log(data.key);
}
```

所以，Mobx 与 Redux 技术选型的背后，其本质是可变数据与不可变数据的选择。不过在项目中，对于两种方案的选择还需要根据实际情况进行分析，脱离业务场景讨论技术选型是没有意义的。

可变数据和不可变数据，都是对状态的一种描述。那么有没有一种方案，能同时作用于可变数据和不可变数据呢？答案是肯定的，它就是 mobx-state-tree（简称 MST），官网地址：<https://github.com/mobxjs/mobx-state-tree>。读者可以自行参考官网资料的介绍。

6.2.6 react-native-art

在移动应用开发过程中，绘制基本的二维图形或动画是必不可少的内容。考虑到 Android 和 iOS 均有一套各自的 API 实现方案，因此采用一种更普遍跨平台技术方案，更有利于多平台兼容而 react-native-art 就是一个不可多得的跨平台技术实现。

art 是一个旨在多浏览器兼容的 Node style CommonJS 模块，在它的基础上，Facebook 将它封装成 react-art，使之可以在 React.JS 中使用。考虑到跨平台的需求，加之前端的技术积累，ReactNative 分别在 0.10.0 和 0.18.0 版本中添加了 iOS 和 Android 平台上对 react-art 的支持。

在 React Native 开发中，ART 是个非常重要的库，它让非常酷炫的绘图及动画变成了可能。在使用 ART 开发前，需要先添加库依赖：Android 默认包含 ART 库，iOS 需要单独添加依赖。iOS 添加依赖的过程如下。

❶ 使用 Xcode 打开 iOS 项目，右键单击【Libraries】，选择【Add Files to…】，依次打开 node_modules/react-native/Libraries/ART/ART.xcodeproj 添加依赖，如图 6-8 所示。

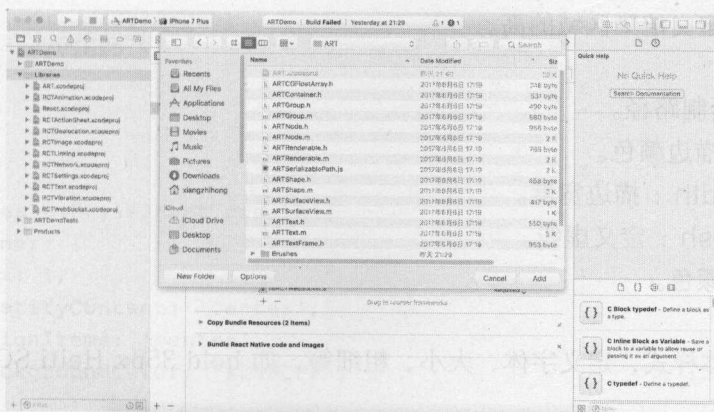


图6-8 添加ART.xcodeproj

② 选中项目根目录，点击【Build Phases】→【Link Binary With Libraries】→+→选中【libART.a】即可，如图6-9所示。

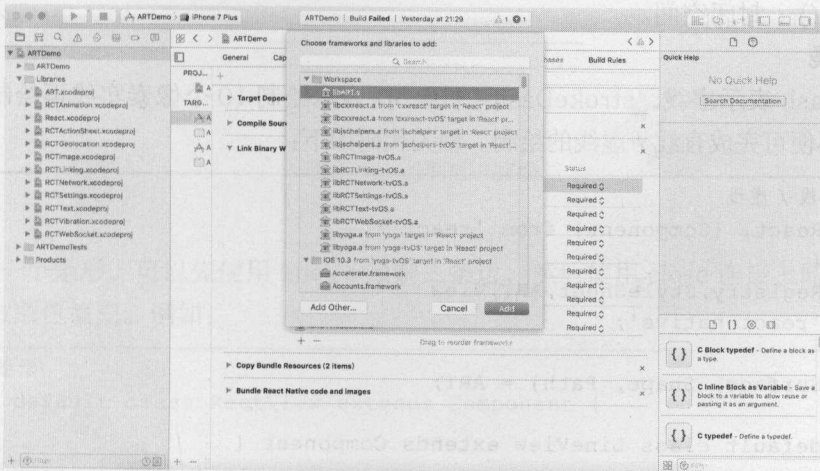


图6-9 关联libART.a

基础组件

ART 暴露的组件共有 7 个，其中常用的有 4 个：Surface、Group、Shape、Text。

- Surface：一个矩形可渲染的区域，是其他元素的容器。
- Group：可容纳多个形状、文本和其他的分组。
- Shape：形状定义，可填充。
- Text：文本形状定义。

其中，Surface、Group、Shape、Text 提供的属性如下。

属性

Surface

- width：渲染区域的宽。

- height : 定义渲染区域的高。

Shape

- d : 定义绘制路径。
- stroke : 描边颜色。
- strokeWidth : 描边宽度。
- strokeDash : 定义虚线。
- fill : 填充颜色。

Text

- font : 字体样式, 定义字体、大小、粗细等, 如 bold 35px Heiti SC。

Path

- moveTo(x,y) : 移动到坐标 (x,y) 。
- lineTo(x,y) : 连线到 (x,y) 。
- arc() : 绘制弧线。
- close() : 封闭空间。

绘制直线/虚线

strokeDash 表示虚线, strokeDash=[10,5] 表示绘制 10 个像素实线再绘制 5 个像素空白, 如此循环便可完成直线 / 虚线的绘制。相关代码如下:

```
// 绘制直线 / 虚线
import React, {Component} from 'react';
import {
  AppRegistry, StyleSheet, ART, View
} from 'react-native';

const {Surface, Shape, Path} = ART;

export default class LineView extends Component {

  render() {

    const path = Path()
      .moveTo(1, 1)
      .lineTo(300, 1);

    return (
      <View style={styles.container}>
        <Surface width={300} height={2}>
          <Shape d={path} stroke="#000000" strokeWidth={2} />
        </Surface>
        <View style={{marginTop:20}}/>
        <Surface width={300} height={2}>
          <Shape d={path} stroke="#000000" strokeWidth={2}
strokeDash={[10, 5]}/>

```



```

        </Surface>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
});

```

运行效果如图 6-10 所示。

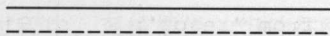


图6-10 绘制直线/虚线

绘制矩形

要绘制一个矩形，可以先使用 `lineTo` 绘制三条边，然后使用 `close` 连接上即可，当然还可以使用 `fill` 做颜色填充。例如：

```

// 绘制矩形
export default class RectView extends Component {

  render() {

    const path = new Path()
    .moveTo(1,1)
    .lineTo(1,99)
    .lineTo(99,99)
    .lineTo(99,1)
    .close();

    return (
      <View style={styles.container}>
        <Surface width={100} height={100}>
          <Shape d={path} stroke="#000000" fill="#892265"
            strokeWidth={1} />
        </Surface>
      </View>
    );
  }
}

```



```

    </View>
  );
}
}

```

运行效果如图 6-11 所示。



图6-11 绘制矩形

绘制文本

使用 ART 的 Text 可以绘制文字，通过设置 Text 的 font 属性可以改变字体的粗细、大小。例如：

```

// 绘制文本
import React, {Component} from 'react';
import {
  AppRegistry,
  StyleSheet,
  ART,
  View
} from 'react-native';

const {Surface, Text, Path} = ART;

export default class ArtTextView extends Component {

  render() {

    return (
      <View style={styles.container}>
        <Surface width={100} height={100}>
          <Text strokeWidth={1} stroke="#000" font="bold 35px
Heiti SC" path={new Path().moveTo(40,40).lineTo(99,10)} >React</Text>
        </Surface>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',

```

```

    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  }
});

```

运行效果如图 6-12 所示。

React

图6-12 绘制文本

除了上述的简单属性之外，还可以使用 Group 绘制叠加图层，并且使用 ART 与动画结合还可以创造出各种绚烂的动画效果。

6.3 自定义组件

在使用组件方式开发应用的过程中，系统提供的组件往往是有限的，亦或者系统提供的组件不能达到开发需求的时候，我们首先会考虑自定义组件。亦或者一些功能相似，可以重用的代码，也可以考虑自定义组件。在使用 React Native 开发项目的过程中除了合理地使用第三方开源库之外，使用自定义组件的场景也是很多的。

6.3.1 组件的导出导入

在 React Native 中，组成界面的最基本元素就是组件，我们可以导入系统提供的组件，也可以自己封装一些组件导出。

❶ 组件的导入导出。示例代码如下：

```

// 导入
import HelloComponent from './HelloComponent'
// 导出
export default HelloComponent

```

❷ 变量的导入导出。示例代码如下：

```

// 变量导出
var name='hello'
export{name }
或者
export var name = 'hello'

// 导入
import HelloComponent, {name } from './HelloComponent'

```

❸ 方法的导入导出。示例代码如下：

```
// 导出
export function sum(a,b){
  return a+b;
}
```

6.3.2 TabbarView封装

在上一节中，使用第三方库 react-native-tab-navigator 可以很轻松地实现页面导航效果，细心的读者可能注意到了，对于 TabNavigator.Item 选项卡部分，代码功能上基本上是重复的。对此，我们能不能对这种有相同功能的代码进行二次封装呢？答案是肯定的。封装不仅可以优化代码的组织结构，也方便项目的后期维护。例如，下面是对常见的 Tabbar 进行模块切换的封装，如图 6-13 所示是 TabbarView 封装 Tab 切换。

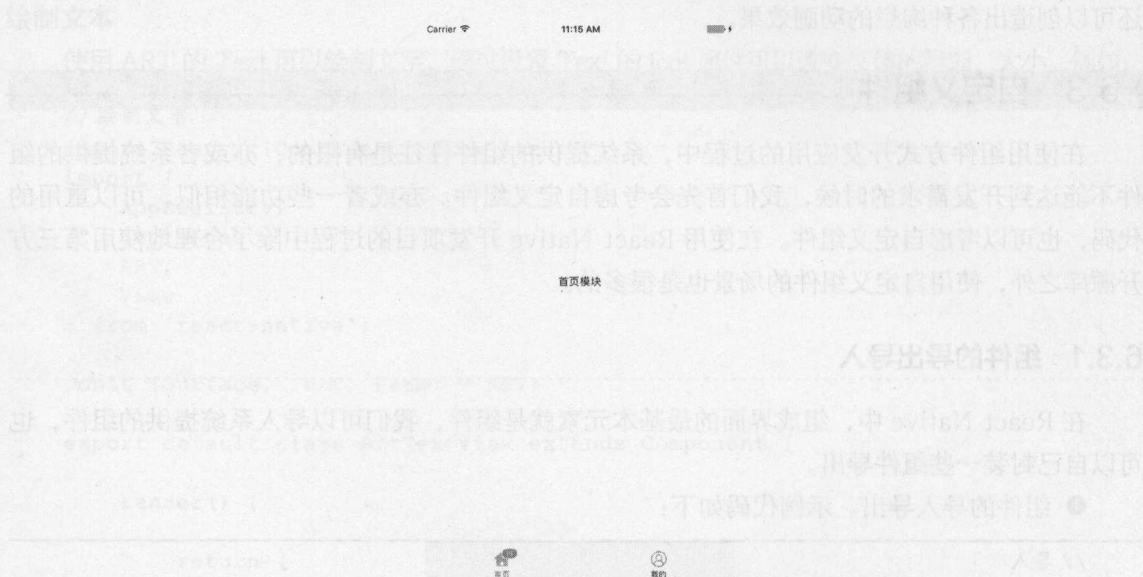


图6-13 TabbarView封装Tab切换

要完整封装 TabbarView，首先需要对 Tab 选项卡进行抽取。Tab 选项卡必备的属性有：选项卡名称、选项卡图标、选项卡对应的视图、选项卡的角标、默认选中的选项卡。

那么，构造函数可以定义如下：

```
// 名称，图标，子视图文本，选中状态
renderTabView(title,tabName,tabContent,isBadge)
```

根据 tabName 参数来区分当前选项卡是否被选中。我们可以对选项卡的图标默认 / 选中状态进行设置：

```
switch (tabName) {
  case 'Home':
```

```

        tabNormal=TAB_HOME_NORMAL;
        tabPress=TAB_HOME_PRESS;
        break;
    case 'Mine':
        tabNormal=TAB_MINE_NORMAL;
        tabPress=TAB_MINE_PRESS;
        break;
    default:
}

```

然后根据构造参数传递的值，对 TabNavigatorItem 选项卡进行渲染：

```

return(
  <TabNavigatorItem
    selected={this.state.selectedTab===tabName}
    title={title}
    titleStyle={styles.tabText}
    selectedTitleStyle={styles.selectedTabText}
    renderIcon={()=><Image style={styles.icon} source={tabNormal}/>}
    renderSelectedIcon={()=><Image style={styles.icon} source={tabPress}/>}

    onPress={()=>this.onPress(tabName)}
    // 角标
    renderBadge={()=>isBadge?<View style={styles.badgeView}><Text style=
{styles.badgeText}>15</Text></View>:null}>
    // 选项卡对应的子 View
    <View style={styles.page}>
      <Text>{tabContent}</Text>
    </View>
  </TabNavigatorItem>
);

```

接下来，在主函数的 render() 方法里面直接按照构造函数设置参数即可：

```

<TabNavigator
  tabBarStyle={styles.tab}>
  {this.renderTabView('首页','Home','首页模块',true)}
  {this.renderTabView('我的','Mine','我的模块',false)}
</TabNavigator>

```

最后，将封装的组件使用 export default 导出。完整实例如下：

```

const TabNavigatorItem = TabNavigator.Item;
// 默认选项
const TAB_HOME_NORMAL=require('./image/tabbar_homepage.png');
const TAB_MINE_NORMAL=require('./image/tabbar_mine.png');
// 选中
const TAB_HOME_PRESS=require('./image/tabbar_homepage_selected.png');
const TAB_MINE_PRESS=require('./image/tabbar_mine_selected.png');

```



```

class TabNavigatorView extends Component {
  // 默认选中
  constructor() {
    super();
    this.state = {
      selectedTab: 'Home',
    }
  }
  // 点击方法
  onPress(tabName) {
    if (tabName) {
      this.setState({
        selectedTab: tabName,
      })
    }
  }
}

// 渲染选项卡
renderTabView(title, tabName, defaultTab, isBadge) {
  var tabNormal;
  var tabPress;
  var tabPage;
  switch (tabName) {
    case 'Home':
      tabNormal = TAB_HOME_NORMAL;
      tabPress = TAB_HOME_PRESS;
      tabPage = <HomeScreen />;
      break;
    case 'Mine':
      tabNormal = TAB_MINE_NORMAL;
      tabPress = TAB_MINE_PRESS;
      tabPage = <MineScreen />;
      break;
    default:
  }

  return (
    <TabNavigatorItem
      selected={this.state.selectedTab===tabName}
      title={title}
      titleStyle={styles.tabText}
      selectedTitleStyle={styles.selectedTabText}
      renderIcon={()=><Image style={styles.icon} source={tabNormal}/>}
      renderSelectedIcon={()=><Image style={styles.icon} source={tabPress}/>}
      onPress={()=>this.onPress(tabName)}
    />
  )
}

```

```

renderBadge={() => isBadge ? <View style={styles.badgeView}><Text style=
{styles.badgeText}>15</Text></View>: null}
    >
    <View style={styles.page}>
      {tabPage}
    </View>
  </TabNavigatorItem>
);
}
// 自定义 TabView
tabBarView() {
  return (
    <TabNavigator
      tabBarStyle={styles.tab}>
      {this.renderTabView(' 首页 ', 'Home', HomeScreen, true)}
      {this.renderTabView(' 我的 ', 'Mine', HomeScreen, false)}
    </TabNavigator>
  );
}

// 渲染界面
render() {
  var tabView = this.tabBarView();
  return (
    <View style={styles.container}>
      {tabView}
    </View>
  );
}
}

```

6.3.3 九宫格布局封装

在移动 APP 中，九宫格是一种很经典的界面布局。在原生 APP 开发中，Android 提供了网格控件 GridView，开发者可以很轻松地实现九宫格界面。而在 iOS 中则需要开发者自己去实现，不过也相对简单。在 React Native 中，系统并没有直接提供相关的控件，但可以通过手动封装，来实现九宫格布局。

要实现自定义九宫格界面，可以使用 ListView 组件来实现。如图 6-14 所示是九宫格界面。



图6-14 自定义九宫格界面

```
// 使用 for 循环绘制九宫格
renderGrid() {
  let data = [];
  let allData=this.loadGridInfos();
  for (var i = 0; i < allData.length; i++) {
    var item = allData[i];
    var itemImage='https://p0.meituan.net/120.0/feop/aa6cff674c96581e502485c55bce4ce039401.png';
    data.push(
      <View key={i} style={styles.gridStyle}>
        <Image style={styles.imageStyle} source={{uri: itemImage}}> </Image>
        <Text>{item.title}</Text>
      </View>
    );
  }
  return data; }

```

在封装组件的时候，为了让封装的控件扩展性更好，开发的时候需要对自定义组件进行合理的抽象，为了达到这一目的。通常采用的方法是：自定义一些属性给调用者使用，当调用者传入不同的属性值即可达到不同的显示效果。

在一些大型 APP 中，经常会看到一种很经典的设计：九宫格 + 指示器，这种扩展的九宫格一般作为二级页面的入口，效果如图 6-15 所示。那么，要实现这种效果，应该怎么实现呢？如图 6-16 所示描述了绘制九宫格界面的流程图。

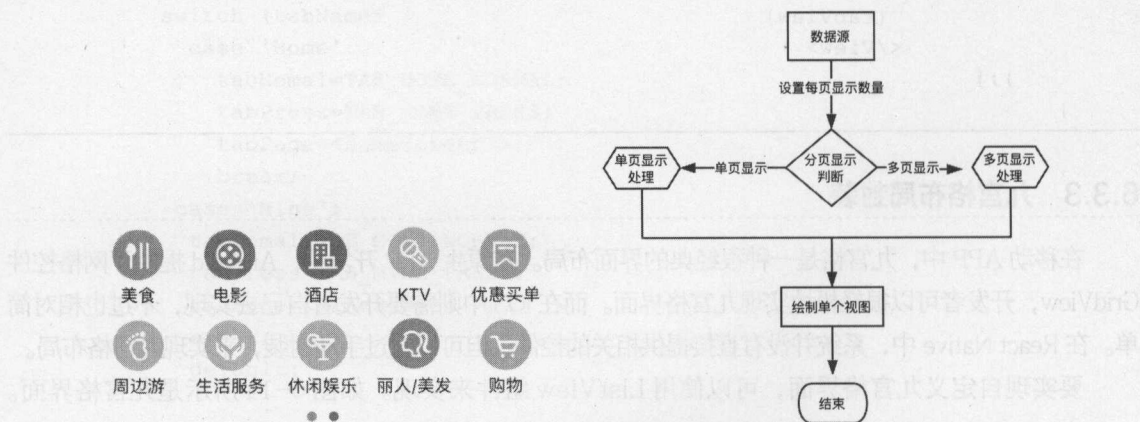


图6-15 复杂的九宫格界面

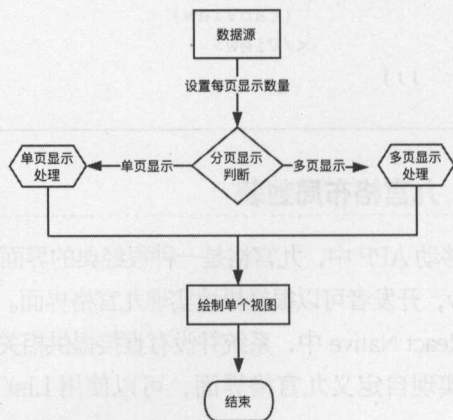


图6-16 绘制九宫格界面流程图

要实现该界面效果，在 Android 中可以使用系统提供的 ViewPagerAndroid，而 iOS 平台似乎没有直接的组件。在 React Native 开发中，系统也没有为开发者提供类似的组件，要实现这种效果，就只能用自定义控件实现。

① 通过返回的数据计算页面数量的代码如下：

```
var pageSize=10;
let menuViews = []
let pageCount = Math.ceil(menuItems.length / pageSize)

```

```

for (let i = 0; i < pageCount; i++) {
  let length = menuItems.length < (i * pageSize) ? menuItems.length - (i *
pageSize) : pageSize
  let items = menuItems.slice(i * pageSize, i * pageSize + length)

  let menuView = (
    <View style={styles.itemsView} key={i}>
      {items}
    </View>
  )
  menuViews.push(menuView)
}

```

② 使用 Scroview 包含需要渲染的 View 视图来渲染界面:

```

<ScrollView contentContainerStyle={styles.contentContainer}
  horizontal={true}
  showsHorizontalScrollIndicator={false}
  pagingEnabled={true}
  onScroll={(e) => this.onScroll(e)}>
  <View style={styles.menuContainer}>
    {menuViews}
  </View>
</ScrollView>

```

③ 用自定义组件来处理页面指示器滚动逻辑（其实应该将这部分逻辑放在界面渲染之前，并且根据实际显示需求可以对参数进行修改）。

```

<PageControl
  style={styles.pageControl}
  numberOfPages={pageCount}
  currentPage={this.state.currentPage}
  hidesForSinglePage={true}
  pageIndicatorTintColor='gray'
  currentPageIndicatorTintColor='#06C1AE'
  indicatorSize={{ width: 8, height: 8 }} />

```

④ PageControl 封装好的用来处理滚动相关逻辑的自定义组件。核心代码如下:

```

class PageControl extends Component {

  onPageIndicatorPress(index: number) {
    this.props.onPageIndicatorPress(index);
  }

  render() {
    var { style, ...props } = this.props;

    var defaultStyle = {
      height: this.props.indicatorSize.height
    }

```



```

    };

    var indicatorItemStyle = {
      width: this.props.indicatorSize.width,
      height: this.props.indicatorSize.height,
      borderRadius: this.props.indicatorSize.height / 2,
      marginLeft: 5,
      marginRight: 5
    };

    var indicatorStyle = assign({}, indicatorItemStyle, this.props.
indicatorStyle, {
      backgroundColor: this.props.pageIndicatorTintColor
    });

    var currentIndicatorStyle = assign({}, indicatorItemStyle, this.
props.currentIndicatorStyle, {
      backgroundColor: this.props.currentPageIndicatorTintColor
    });

    var pages = [];
    for (var i = 0; i < this.props.numberOfPages; i++) {
      pages.push(i);
    }

    return (
      this.props.hidesForSinglePage && pages.length <= 1 ? null :
<View style={[styles.container, defaultStyle, style]}>
      {pages.map((el, i) => <TouchableWithoutFeedback key={i}
onPress={this.onPageIndicatorPress.bind(this, i)}>
        <View style={i === this.props.currentPage ? currentIndicatorStyle :
indicatorStyle} />
        </TouchableWithoutFeedback>
      )}
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    backgroundColor: 'transparent',
    alignItems: 'center',
    justifyContent: 'center',
    flexDirection: 'row'
  }
});

export default PageControl;

```

关于本部分示例的完整代码，请查看随书源码。

6.3.4 下拉刷新组件封装

下拉刷新作为移动开发的基本功能组件，经常用在上拉刷新和下拉加载更多中。而到目前为止，比较好用的下拉开源库几乎很难找到，而有些提供这样功能的库使用上也比较复杂，例如 react-native-refreshable-listview。所以，有必要使用 ListView 和 RefreshControl 封装一个下拉刷新组件。

首先，简单介绍下 RefreshControl。

RefreshControl属性

RefreshControl 提供的属性如下。

- refreshing
加载指示器是否在加载中。
- colors [ColorPropType]
Android平台专用，设置加载指示器的颜色，至少设置一种，最多可以设置4种。
- enabled
Android平台专用，用来设置下拉刷新是否可用。
- progressBackgroundColor [ColorPropType]
用来设置加载指示器的背景颜色。
- size
Android平台专用，用来设置加载指示器的尺寸大小。默认值为RefreshLayout Consts.SIZE.DEFAULT。
- tintColor [ColorPropType]
iOS平台专用，用来设置加载指示器的颜色。
- title
iOS平台专用，用来设置加载指示器下面的标题文本信息。

RefreshControl方法

- onRefresh()

当视图开始刷新的时候，调用对于下拉过程中加载的效果需要使用一个新的组件 ActivityIndicator，它是 React Native 提供的指示器，提供加载过程中旋转的效果。默认有 4 个属性：size、color、animating 和 hidesWhenStopped。

- size: 指示器大小，提供的枚举值，即large、small，默认是small。在Android平台支持设置指定数字大小。
- color: 指示器颜色，默认为white。
- animating: 是否显示动画指示器，默认是显示true。
- hidesWhenStopped: 在没有动画指示器的时候是否隐藏，默认为true。

RefreshListView封装

- ❶ 使用 import 导入相关系统组件代码如下：

```
import {ListView,RefreshControl,ActivityIndicator} from 'react-native';
```

- ❷ 定义 RefreshListView 的各种状态：

```
const RefreshState = {
  Idle: 'Idle',
  Refreshing: 'Refreshing',
  NoMoreData: 'NoMoreData',
  Failure: 'Failure'
}
```

- ❸ 使用 RefreshControl 组件来实现下拉刷新，即通过定义的刷新方法和属性来控制当前组件的状态，从而给出相应的反馈结果：

```
refreshControl={
  <RefreshControl
    refreshing={this.state.headerState == RefreshState.Refreshing}
    onRefresh={() => this.onHeaderRefresh()}
    tint color='gray'
  />
}
```

当上拉的时候，触发上拉操作，执行上拉加载动画界面渲染，当滚动到底部时触发 onEndReached 方法，执行 Footer 文字描述界面渲染，并通过 RefreshState 的相关状态来改变默认参数的值。核心代码如下：

```
renderFooter() {
  let footer = null;
  switch (this.state.footerState) {
    case RefreshState.Idle:
      break;
    case RefreshState.Failure: {
      footer =
        <TouchableOpacity style={styles.footerContainer}
          onPress={() => this.startFooterRefreshing()}
        >
          <Text style={styles.footerText}>
            {this.props.footerFailureText}
          </Text>
        </TouchableOpacity>
      break;
    }
    case RefreshState.Refreshing: {
      footer =
        <View style={styles.footerContainer} >
          <ActivityIndicator size="small" color="#888888" />
        </View>
      break;
    }
  }
}
```

```

        <Text style={styles.footerText}>
          {this.props.footerRefreshingText}
        </Text>
      </View>
      break;
    }
    case RefreshState.NoMoreData: {
      footer =
        <View style={styles.footerContainer} >
          <Text style={styles.footerText}>
            {this.props.footerNoMoreDataText}
          </Text>
        </View>
      break;
    }
  }
  return footer;
}

```

然后，在 render 函数里面调用 ListView 组件来执行界面绘制，页面显示相关的逻辑由 RefreshControl 控制。相关代码如下：

```

render() {
  return (
    <ListView
      {...this.props}
      enableEmptySections
      refreshControl={
        <RefreshControl
          refreshing={this.state.headerState == RefreshState.Refreshing}
          onRefresh={() => this.onHeaderRefresh()}
          tint='gray'
        />
      }
      renderFooter={() => this.renderFooter()}
      onEndReachedThreshold={10}
      onEndReached={() => this.onFooterRefresh()} />
  );
}

```

④ RefreshListView.js 下拉刷新组件完整代码如下：

```

import React, { Component } from 'react';
import { View, Text, StyleSheet, RefreshControl, ListView, ActivityIndicator,
TouchableOpacity } from 'react-native';

export const RefreshState = {
  Idle: 'Idle',

```



```

    Refreshing: 'Refreshing',
    NoMoreData: 'NoMoreData',
    Failure: 'Failure'
  }

export default class RefreshListView extends Component {

  static propTypes = {
    onHeaderRefresh: React.PropTypes.func,
    onFooterRefresh: React.PropTypes.func,
  }

  static defaultProps = {
    footerRefreshingText: '数据加载中……',
    footerFailureText: '点击重新加载',
    footerNoMoreDataText: '已加载全部数据'
  }

  constructor(props: Object) {
    super(props)
    this.state = {
      headerState: RefreshState.Idle,
      footerState: RefreshState.Idle,
    }
  }

  startHeaderRefreshing() {
    this.setState({ headerState: RefreshState.Refreshing })
    if (this.props.onHeaderRefresh) {
      this.props.onHeaderRefresh()
    }
  }

  startFooterRefreshing() {
    this.setState({ footerState: RefreshState.Refreshing })
    if (this.props.onFooterRefresh) {
      this.props.onFooterRefresh()
    }
  }

  shouldStartHeaderRefreshing() {
    if (this.state.headerState == RefreshState.Refreshing ||
        this.state.footerState == RefreshState.Refreshing) {
      return false
    }
    return true
  }
}

```

```

shouldStartFooterRefreshing() {
  if (this.state.headerState == RefreshState.Refreshing ||
      this.state.footerState == RefreshState.Refreshing) {
    return false
  }
  if (this.state.footerState == RefreshState.Failure ||
      this.state.footerState == RefreshState.NoMoreData) {
    return false
  }
  if (this.props.dataSource.getRowCount() == 0) {
    return false
  }
  return true
}

endRefreshing(refreshState: RefreshState) {
  if (refreshState == RefreshState.Refreshing) {
    return
  }
  let footerState = refreshState
  if (this.props.dataSource.getRowCount() == 0) {
    footerState = RefreshState.Idle
  }

  this.setState({
    headerState: RefreshState.Idle,
    footerState: footerState
  })
}

headerState() {
  return self.state.headerState
}

footerState() {
  return self.state.footerState
}

onHeaderRefresh() {
  if (this.shouldStartHeaderRefreshing()) {
    this.startHeaderRefreshing();
  }
}

onFooterRefresh() {
  if (this.shouldStartFooterRefreshing()) {
    this.startFooterRefreshing();
  }
}

```

```

    }

    render() {
      return (
        <ListView
          {...this.props}
          enableEmptySections
          refreshControl={
            <RefreshControl
              refreshing={this.state.headerState === RefreshState.Refreshing}
              onRefresh={() => this.onHeaderRefresh()}
              tint='gray'
            />
          }
        />
        <renderFooter={() => this.renderFooter()}
          onEndReachedThreshold={10}
          onEndReached={() => this.onFooterRefresh()}
        />
      );
    }
  }

```

```

renderFooter() {
  let footer = null;

  switch (this.state.footerState) {
    case RefreshState.Idle:
      break;
    case RefreshState.Failure: {
      footer =
        <TouchableOpacity style={styles.footerContainer}
          onPress={() => this.startFooterRefreshing()}
        >
          <Text style={styles.footerText}>
            {this.props.footerFailureText}
          </Text>
        </TouchableOpacity>
      break;
    }
    case RefreshState.Refreshing: {
      footer =
        <View style={styles.footerContainer} >
          <ActivityIndicator size="small" color="#888888" />
          <Text style={styles.footerText}>
            {this.props.footerRefreshingText}
          </Text>
        </View>
      break;
    }
  }
}

```

```

    case RefreshState.NoMoreData: {
      footer =
        <View style={styles.footerContainer} >
          <Text style={styles.footerText}>
            {this.props.footerNoMoreDataText}
          </Text>
        </View>
      break;
    }
  }
  return footer;
}

const styles = StyleSheet.create({
  footerContainer: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'center',
    alignItems: 'center',
    padding: 10
  },
  footerText: {
    fontSize: 14,
    color: '#555555'
  }
});

```

到此，RefreshListView 自定义组件就封装完成了，下面结合示例介绍下使用方法。

RefreshListView示例

在很多 APP 中都会涉及下拉刷新和上拉加载功能。借助自定义组件 RefreshListView，如何实现图 6-17 的效果呢？



图6-17 RefreshListView使用

❶ 首先，导入 RefreshListView 组件，并设置相关参数和属性：

```

<RefreshListView
  ref='listView'
  dataSource={this.state.dataSource}
  // 渲染 Cell
  renderRow={(rowData) =>

```



```

    <RecommendCell
      info={rowData}
    />
  }
  // 下拉刷新整个视图
  onHeaderRefresh={() => this.requestData()} />

```

② 然后，使用 fetch 函数请求数据并填充界面：

```

requestData () {
  fetch(api.recommend)
    .then((response) => response.json())
    .then((json) => {
      // 数据转换
      let dataList = json.data.map((info) => {
        return {
          id: info.id,
          imageUrl: info.squareimgurl,
          title: info.mname,
          subtitle: '[$${info.range}]$${info.title}',
          price: info.price
        }
      })
      // 设置更新数据源
      this.setState({
        dataSource: this.state.dataSource.cloneWithRows(dataList)
      })
      // 设置加载完全提示
      setTimeout(() => {
        this.refs.listView.endRefreshing(RefreshState.NoMoreData)
      }, 500);
    })
    .catch((error) => {
      this.refs.listView.endRefreshing(RefreshState.Failure)
    })
  }
}

```

当数据返回之后，就可以通过 Promise 机制通知界面刷新，还可以在 fetch 方法里面对错误进行处理。关于这部分的知识，读者可以通过查看随书源码来学习。

6.4 小结

在本章中，通过对组件生命周期的介绍，让我们可以更加深刻地理解组件从创建到销毁的过程，精准掌握并熟练使用生命周期函数会让我们的代码更有逻辑性，通过对常用第三方库的介绍，可以大大提高开发的效率，同时合理使用第三方组件也可以提升应用的性能和体验。而通过自定义组件，可以加深我们对 React Native 原理的理解，同时自定义组件也可以解耦合，加强代码的可维护性。

网络与通信

通信是移动开发的重要部分，学习了解 Native 与 JavaScript 之间的通信，对于深入理解 React Native 设计原理是很有帮助的。而在应用层面上，学习本章，读者将了解到客户端与服务器端网络数据交互相关的内容。学习本章，重点掌握 React Native 在 Native 和 JavaScript 本地通信原理，以及基于 fetch 方式的网络数据请求。

7.1 通信机制

在 React Native 设计之初，Facebook 希望所编写的 JS 代码可以在各个平台上运行，同时又让应用具有原生应用的体验。不过，对于没有从事过原生开发的人来说，React Native 是如何将 JS 代码转换成原生系统能够执行的代码的呢？又是如何与原生系统进行通信的呢？为了弄清这些问题，我们需要理解 React Native 在和 Android、iOS 等平台究竟是如何进行通信的。

Native 与 JS 通信无非就是 Java/OC 与 JS 跨语言间的调用。调用通常会涉及参数传递、方法互调等问题。所以，在谈 React Native 和原生系统进行通信的时候，就会涉及和 Android/iOS 系统之间的通信，如图 7-1 展示了 React Native 整体框架。

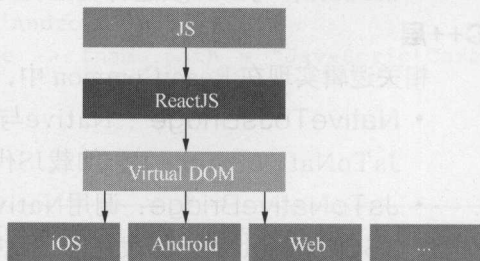


图 7-1 React Native 整体框架

7.1.1 React Native与Android通信

对于传统 Java 和 JS 通信而言, JS 调用 Java 不外乎有 JSbridge、onprompt、log 以及 addjavascript tinterface 四种方式。在 Java 调用 JS 的过程中, 只有 loadurl 和高版本才支持 evaluateJavaScript。但在 React Native 中, 并没有采用传统的 Java 与 JS 之间的通信机制, 而是借助 MessageQueue 及模块配置表, 将调用转化为 {moduleID, methodID, callbackID, args}, 处理端通过在模块配置表里查找注册的模块与方法来实现通信。

其实, 在 Android 系统中, 关于 Java 与 JS 之间的调用早已有了实现, 这就是 WebView。WebView 的底层实现其实就是 WebKit, 尽管在 Android 4.4 版本之后切换成了 Chromium, 但归根结底其底层实现还是使用的 WebKit, WebKit 主要包括 WebCore 排版引擎和 JSCore 引擎。不过 React Native 并没有使用到 WebKit 相关的内容, 而是直接使用 JSCore 引擎来完成界面的渲染, 页面排版则交给 Native 代码去完成。

为了弄清 Android 和 React Native 的通信流程, 首先未完成界面的渲染。

如图 7-2 所示, 整个 React Native 和 JS 通信的框架图主要由 3 部分组成: Java 层、C++层和 JS 层。各个层次的功能和作用如下。

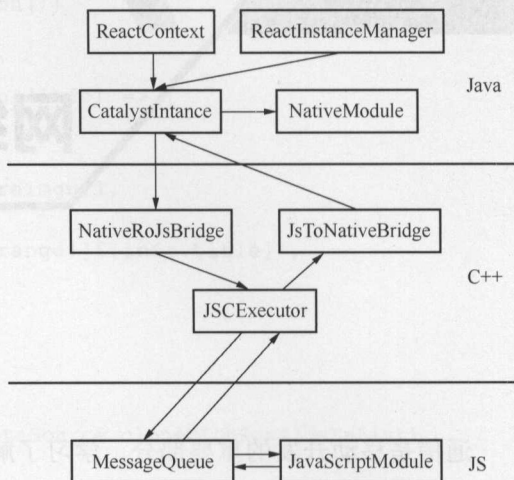


图7-2 React Native通信框架图

Java层

相关逻辑实现在 ReactAndroid 中。涉及的内容如下。

- **ReactContext**: Android上下文子类, 包含一个CatalystInstance实例, 用于获取NativeModule和JSModule、添加各种回调、处理异常等。
- **ReactInstanceManager**: 管理CatalystInstance的实例, 处理RN Root View, 启动JS页面, 管理生命周期。
- **CatalystInstance**: 通信的关键类, 提供调用JS Module, 也支持JS调用Native Module, 与Bridge进行交互, 对开发者不可见。

C++层

相关逻辑实现在 ReactCommon 中, 涉及的内容如下:

- **NativeToJsBridge**: Native与JS的桥梁, 负责调用JS Module, 回调Native (调用JsToNativeBridge), 加载JS代码 (调用JavaScriptCore)。
- **JsToNativeBridge**: 调用Native Module的方法。
- **JSCExecutor**: 加载执行JS代码 (调用JavaScriptCore), 调用JS Module, 回调native, 性能统计等, 都是比较核心的功能。

JS层

主要的实现逻辑在 Libraries 文件中，与 React Native 相关的 JS 实现都可以在这一层找到。涉及的内容如下。

- MessageQueue : 管理JS的调用队列、调用Native/JS Module的方法、执行callback、管理JS Module等。
- JavaScriptModule : 所有的组件都必须继承JSMModule，并在CatalystInstance中注册。

在Native与JS的通信框架中，Java层只做接口定义，而实现则由Javasurript完成。

C++与JS通信

在 React Native 中，JavaScriptCore 用来执行 JS 相关的逻辑，那么 React Native 是如何利用 JavaScriptCore 的呢？相关代码在 Android 编译脚本文件中。

```
compile 'org.webkit:android-jsc:r174650'

task downloadJSCHeaders(type: Download) {
    def jscAPIBaseURL = 'https://svn.webkit.org/repository/webkit/ !svn/
bc/174650/trunk/Source/JavaScriptCore/API/'
    def jscHeaderFiles = ['JavaScript.h', 'JSBase.h', 'JSContextRef.
h', 'JSObjectRef.h', 'JSRetainPtr.h', 'JSStringRef.h', 'JSValueRef.h',
'WebKitAvailability.h']
    def output = new File(downloadsDir, 'jsc')
    output.mkdirs()
    src(jscHeaderFiles.collect { headerName -> "$jscAPIBaseURL$headerName"})
    onlyIfNewer true
    overwrite false
    dest output
}

// Create Android.mk library module based on so files from mvn + include
headers fetched from webkit .org
task prepareJSC(dependsOn: downloadJSCHeaders) << {
    copy {
        from zipTree(configurations.compile.fileCollection { dep -> dep.
name == 'android-jsc' }). singleFile()
        from {downloadJSCHeaders.dest}
        from 'src/main/jni/third-party/jsc/Android.mk'
        include 'jni/**/*.so', '*.h', 'Android.mk'
        filesMatching('*.h', { fname -> fname.path = "JavaScriptCore/
${fname.path}")})
        into "$thirdPartyNdkDir/jsc";
    }
}
```

从上面的源码可以看出，React Native 并没有用系统自带的 Webkit，而是使用了 JavaScript Core，界面的渲染交由 Native 去实现。

Module Registry

在 React Native 中，在应用启动时，ReactPackage 会自动生成两份模块配置表：NativeModule Registry 及 JavaScriptModuleRegistry。Java 端与 JS 端持有相同的模块配置表，分别 Native 模块或 JS 模块，最终将实例通过 ReactPackage 添加到 createNativeModules 方法中。在 CoreModulesPackage.java 文件中，有如下代码：

```
@Override
public List<NativeModule> createNativeModules(
    ReactApplicationContext catalystApplicationContext) {
    return Arrays.<NativeModule>asList(
        new AndroidInfoModule(),
        new DeviceEventManagerModule(catalystApplicationContext,
mHardwareBackBtnHandler),
        new DebugComponentOwnershipModule(catalystApplicationContext));
    }
    @Override
    public List<Class<? extends JavaScriptModule>> createJSMODULES() {
        return Arrays.asList(
            DeviceEventManagerModule.RCTDeviceEventEmitter.class,
            JSTimersExecution.class,
            RCTEventEmitter.class,
            RCTNativeAppEventEmitter.class,
            AppRegistry.class
        )
    }
    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
        return new ArrayList<>(0);
    }
}
```

JS 模块继承自 JavaScriptModule，并映射在相对应 JS 模块上，通过动态代理实现对 JS 模块的调用。接下来，来看一下和 JavascriptModule 相关的 AppRegistry 代码实现。

```
public interface AppRegistry extends JavaScriptModule {
    void runApplication(String appKey, WritableMap appParameters);
    void unmountApplicationComponentAtRootTag(int rootNodeTag);
}
```

在 AppRegistry.java 文件中，AppRegistry 在加载完 JSbundle 后，Native 就会去启动应用。其中，appKey 为应用的 ID。映射后的每个 JavaScriptModule 的信息保存在 JavaScriptModuleRegistration 中，由 JavaScriptModuleRegistry 统一管理。

```
AndroidInfoModule extends BaseJavaModule {
    @Override
    public String getName() {
        return "AndroidConstants";
    }
    @Override
```

```

public @Nullable Map<String, Object> getConstants() {
    HashMap<String, Object> constants = new HashMap<String, Object>();
    constants.put("Version", Build.VERSION.SDK_INT);
    return constants;
}

```

Java 模块继承自 BaseJavaModule，在 JS 层存在着同名文件用来调用 Native。通过重写 getName 函数，我们可以识别 JS 的模块名，通过注解 @ReactMethod 可识别供 JS 调用的 API 接口。所有 Java 层提供的模块接口由 NativeModuleRegistry 统一对外暴露。

Native与JS通信

Native 与 JS 通信首先需要加载 Bundle 文件，这个过程是在 Native 初始化的时候完成的，Bundle 文件的位置是可配置的。关于 Bundle 文件加载的流程可以查看如图 7-3 所示的流程图。

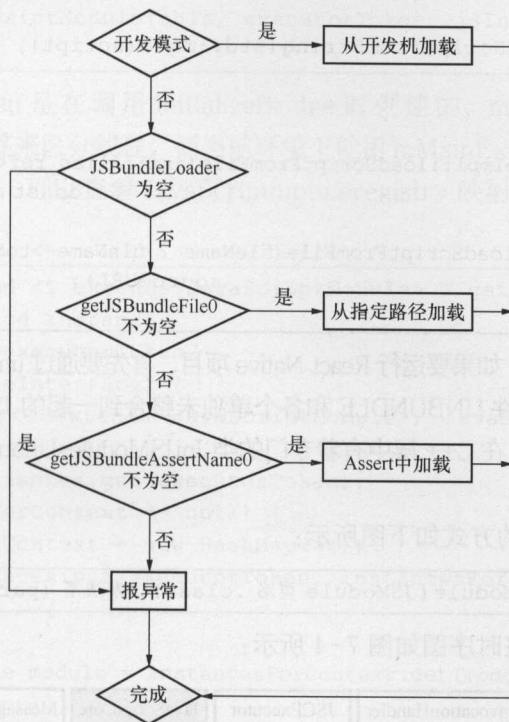


图7-3 Bundle加载流程图

由流程图可知，JSBundleLoader 根据文件的位置去加载相应的 Bundle 文件，最终调用 CatalystIntance 去执行加载，具体实现如下：

```

native void loadScriptFromAssets(AssetManager assetManager, String assetURL);
native void loadScriptFromFile(String fileName, String sourceURL);
native void loadScriptFromOptimizedBundle(String path, String sourceURL, int flags);

```

其中 loadScriptFromOptimizedBundle 支持加载优化后的 Bundle，目前没有用到。对于

从 assets 和文件中加载 Bundle 文件，可以看一下源码实现。

```
void CatalystInstanceImpl::loadScriptFromAssets(jobject assetManager,
                                              const std::string&
assetURL) {
    const int kAssetsLength = 9; // strlen("assets://");
    auto sourceURL = assetURL.substr(kAssetsLength);

    auto manager = react::extractAssetManager(assetManager);
    auto script = react::loadScriptFromAssets(manager, sourceURL);
    if (JniJSMODULESUnbundle::isUnbundle(manager, sourceURL)) {
        instance_>loadUnbundle(
            folly::make_unique<JniJSMODULESUnbundle>(manager, sourceURL),
            std::move(script),
            sourceURL);
        return;
    } else {
        instance_>loadScriptFromString(std::move(script), sourceURL);
    }
}

void CatalystInstanceImpl::loadScriptFromFile(jni::alias_ref<jstring> fileName,
                                              const std::string&
sourceURL) {
    return instance_>loadScriptFromFile(fileName ? fileName->toStdString() : "",
                                         sourceURL);
}
```

对于 Android 系统而言，如果要运行 React Native 项目，首先要通过 `unbundle` 命令生成 `js-modules` 文件夹，里面存放着标志文件 `UNBUNDLE` 和各个单独未整合到一起的 JS 文件。而 `JSBundleLoader` 在读取 Bundle 文件的时候，在 C++ 层中有着专门的类 `JniJSMODULESUnbundle` 来处理这些文件。

Native调用JS

在 Native 中调用 JS 的方式如下图所示：

```
ReactContext.getJSMODULE(JSModule 类名.class).方法名(params);
```

Native 调用 JS 的完整时序图如图 7-4 所示：

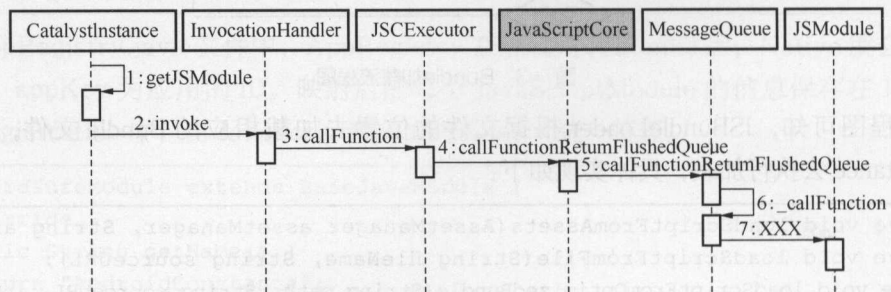


图7-4 Native调用JS时序图

getJSMModule 相关方法的源码如下:

```

@Override
    public <T extends
//rnmmainExeautortoken 来自于原生 C++ 代码实现
JavaScriptModule> T getJSMModule(Class<T> jsInterface) {
    return getJSMModule(mMainExecutorToken, jsInterface);
}

@Override
    public <T extends JavaScriptModule> T getJSMModule(ExecutorToken
executorToken, Class<T> jsInterface) {
    //mjsmodleregistry 就是启动流程中的 processpacrage() 方法, 然后将其交给 Catalystins
tancelmpl 托管的 JS 映射表
    return Assertions.assertNotNull(mJSMModuleRegistry)
        .getJavaScriptModule(this, executorToken, jsInterface);
}

```

mMainExecutorToken 是在调用 initializeBridge 时创建的, mMainExecutorToken 与 JS 多线程相关, 使用某些标签来区分线程, 而当前环境下使用 mMainExecutorToken 就可以。关于 getJSMModule 的具体实现, 可以查看 JavaScriptmotuleregistry 映射来的 getJavaScriptModule 方法相关的代码实现:

```

public synchronized <T extends JavaScriptModule> T getJavaScriptModule(
    CatalystInstance instance,
    ExecutorToken executorToken,
    Class<T> moduleInterface) {
    HashMap<Class<? extends JavaScriptModule>, JavaScriptModule> instances
ForContext =
        mModuleInstances.get(executorToken);
    if (instancesForContext == null) {
        instancesForContext = new HashMap<>();
        mModuleInstances.put(executorToken, instancesForContext);
    }

    JavaScriptModule module = instancesForContext.get(moduleInterface);
    if (module != null) {
        return (T) module;
    }

    JavaScriptModuleRegistration registration =
        Assertions.assertNotNull(
            mModuleRegistrations.get(moduleInterface),
            "JS module " + moduleInterface.getSimpleName() + " hasn't been registered!");
    JavaScriptModule interfaceProxy = (JavaScriptModule) Proxy.newProxyInstance(
        moduleInterface.getClassLoader(),
        new Class[] {moduleInterface},

```

```

        new JavaScriptModuleInvocationHandler(executorToken, instance, registration));
    instancesForContext.put(moduleInterface, interfaceProxy);
    return (T) interfaceProxy;
}

```

在创建 CatalystInstance 的时候，系统会将打包的所有 JavaScriptModule 收集到 JavaScriptModule Registry 的 Map(mModuleRegistrations) 中。mModuleInstances 用于缓存已经调用过的 JavaScript Module 的代理对象，如果已经被调用过，则从 Map 中直接返回，否则创建其代理对象，然后将 JavaScriptModule 缓存起来。

Java 端通过 getJSModule 获取 JSModule 的实质是通过 Java 的动态代理实现的，这种动态代理方式是，先创建一个 interface 代理对象，当调用其方法时会调用 Invocation Handler 会调用 invoke 方法。invoke 方法相关代码如下：

```

public @Nullable Object invoke(Object proxy, Method method, @Nullable
Object[] args) throws
    Throwable {
    ExecutorToken executorToken = mExecutorToken.get();
    if (executorToken == null) {
        FLog.w(ReactConstants.TAG, "Dropping JS call, ExecutorToken went away...");
        return null;
    }
    NativeArray jsArgs = args != null ? Arguments.fromJavaArgs(args) : new
WritableNativeArray();
    mCatalystInstance.callFunction(
        executorToken,
        mModuleRegistration.getName(),
        method.getName(),
        jsArgs
    );
    return null;
}

```

❶ CatalystInstanceImpl 封装了 JS 和 Java 通信的实现类，业务模块通过 ReactInstance Manager 与 CatalystInstanceImpl 间接通信，调用 JS 暴露出来的 API。

❷ 将来自 Java 端的调用拆分为 ModuleID、MethodID 及 Params。JavaScript ModuleInvocation Handler 通过动态代理交由 CatalystInstanceImpl 进行统一管理。

❸ CatalystInstanceImpl 将 ModuleID、MethodID 及 Params 转交给 ReactBridge JNI 进行处理。

❹ ReactBridge 调用 C++ 层，调用链转发 ModuleID、MethodID 及 Params。

❺ JSCHelper 的 evaluateScript() 将 ModuleID、MethodID 及 Params 借助 JSC 传递给 JS 端，进而完成 Java 对 JS 端的调用。

Java 调用 JS 的完整流程如图 7-5 所示。

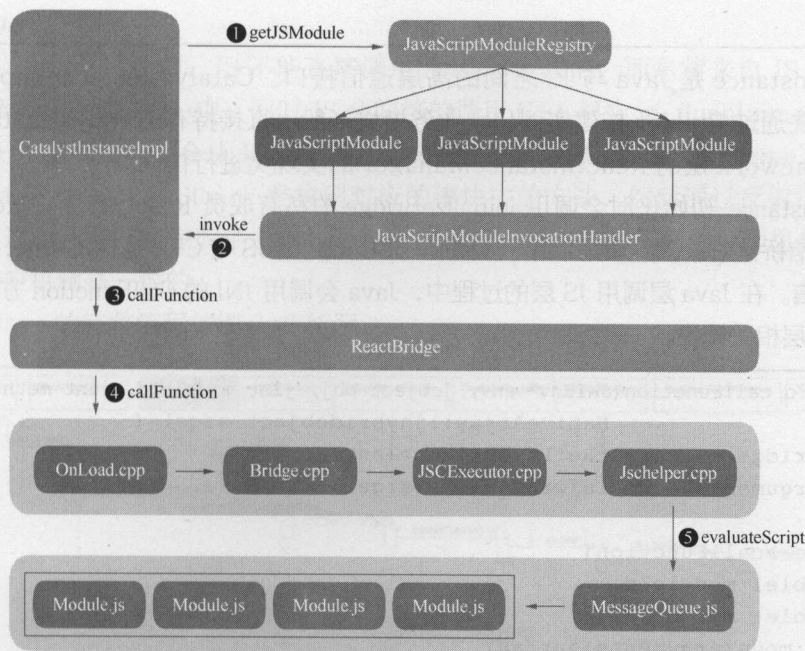


图7-5 Java调用Js流程图

CatalystInstanceImpl 封装了 JS 和 Java 高层通信，通过实现 JavaScriptModule 从而暴露公共的 Module。JavaScriptModuleRegistry 负责管理所有的 JavaScriptModule，持有对 JavaScriptModule InvocationHandler 的引用，通过 invoke 方式，统一调度 Java 到 JS 的调用。

```

private static class JavaScriptModuleInvocationHandler implements InvocationHandler {

    private final CatalystInstanceImpl mCatalystInstance;
    private final JavaScriptModuleRegistration mModuleRegistration;
    public JavaScriptModuleInvocationHandler(
        CatalystInstanceImpl catalystInstance,
        JavaScriptModuleRegistration moduleRegistration) {
        mCatalystInstance = catalystInstance;
        mModuleRegistration = moduleRegistration;
    }

    @Override
    public @Nullable Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        String tracingName = mModuleRegistration.getTracingName(method);
        mCatalystInstance.callFunction(
            mModuleRegistration.getModuleId(),
            mModuleRegistration.getMethodId(method),
            Arguments.fromJavaArgs(args),
            tracingName);
        return null;
    }
}

```

```
}}
```

CatalystInstance 是 Java 与 JS 之间的高层通信接口。CatalystInstanceImpl 为其基础实现类。业务模块通过 Builder 构建实例化，业务模块一般不直接持有 CatalystInstance 的引用，而是通过 Framework 层的 ReactInstanceManager 的实现类进行访问。

CatalystInstance 初始化时会调用 initializeBridge 的私有成员 ReactBridge。ReactBridge 作为 JNI 层的通信桥接对象，实质是依赖于 webkit JSC 架起的 JS 与 C++ 通信的桥架，负责 Java 与 JCS 之间的通信。在 Java 层调用 JS 层的过程中，Java 会调用 JNI 的 CallFunction 方法，然后通过 JSC 转接到 JS 层相关模块。

```
static void callFunction(JNIEnv* env, jobject obj, jint moduleId, jint methodId,
                        NativeArray::jhybridobject args) {
    auto bridge = extractRefPtr<Bridge>(env, obj);
    auto arguments = cthis(wrap_alias(args));
    try {
        bridge->callFunction(
            (double) moduleId,
            (double) methodId,
            std::move(arguments->array)
        );
    } catch (...) {
        translatePendingCppExceptionToJavaException();
    }
}
```

Onload.cpp 为 C++ 层主要入口，涵盖类型操作、Jsbundle 加载及全局变量操作等。通过 bridge.cpp 转接到 JSExector.cpp 中执行 JS。JSExector.cpp 最终将调用转发到 JSCHelper.cpp 中执行 evaluateScript 函数，最终执行 JS 调用。

至此，Java 层调用 C++ 层结束，JSC 将执行 JS 调用。当 JS 层接收来自 C++ 层调用时 JS 端的 MessageQueue。JS 会调用 callFunctionReturnFlushedQueue 函数，最终调用 CallFunction 函数执行 JS 后调用 flushedQueue 更新队列。

```
callFunctionReturnFlushedQueue(module, method, args) {
    guard(() => {
        this.__callFunction(module, method, args);
        this.__callImmediates();
    });
    return this.flushedQueue();
}
```

到此，再来看一下 Ncative 层（即 Java 层）调用 JS 层的完整流程：Java 层和 JS 层都准备好一个 Module 映射表，当 Java 端调用 JS 端方法时，Java 端通过查找映射表，动态创建一个与 JS 端对应的 Module 对象。当调用这个方法时，Java 端通过动态代理的 invoke 方法触发 C++ 层，C++ 层通过调用 JSCEXEcator 执行 JS 端队列中的映射表查找，并找到 JS 端的方法进行调用，同时给 Java 端一个回调。

JS调用Native

在 React Native 设计中, JS 不能直接调用 Native 方法的, 而是将来自 JS 层的调用先推送到 JS 层的 MessageQueue 中, 同时 JS 会间接的调用 C++ 层中 m_flushImmediateCallback 函数, 当 C++ 收到调用后, 会从 JS 传递过来的数据中解析出 moduleName、参数等信息, 然后调用 Java 层的 ReactCallback 类找到对应的模块中的方法, 然后通过反射执行这些方法, 并把参数传递过去。这样就完成了 JS 对 Native 的调用。这种基于队列的处理思想和 Native 开发中的事件响应机制是一致的。

JS 调用 Java 的完整流程如图 7-6 所示。

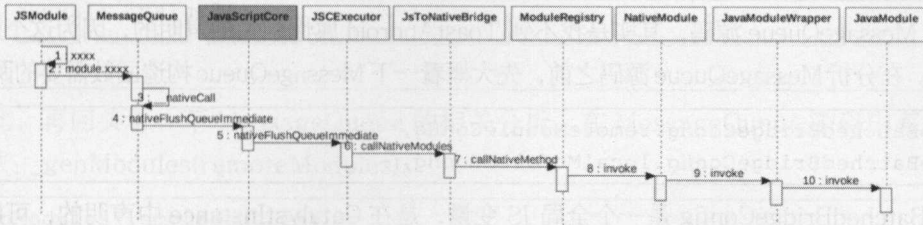


图7-6 JS调用Java流程图

为了方便讲解, 本部分以 React Native 官方提供的《Native Modules》(官方地址: <http://facebook.github.io/react-native/docs/native-modules-android.html>) 为例, 使用大家都比较熟悉的 ToastAndroid 组件来讲解 JS 调用 Native 的过程。首先来看一下 ToastAndroid 组件的实现:

```

var RCTToastAndroid = require('NativeModules').ToastAndroid;
//...
var ToastAndroid = {
  ...
  show: function (
    message: string,
    duration: number
  ): void {
    RCTToastAndroid.show(message, duration);
  },
  //...
};

```

可以看到, show 方法调用了 RCTToastAndroid.show(message, duration) 方法, 而 RCTToastAndroid 又是定义在 NativeModules.js 中的。

```

var NativeModules = require('BatchedBridge').RemoteModules;
var nativeModulePrefixNormalizer = require('nativeModulePrefixNormalizer');
nativeModulePrefixNormalizer(NativeModules);
module.exports = NativeModules;

```

最终, 所有的内容都来自于 BatchedBridge.js, 而 BatchedBridge 做的最重要的一件事就是

构造一个 MessageQueue 对象。所以，ToastAndroid.show() 方法最终调用的是 MessageQueue.ToastAndroid.show() 方法。相关代码如下：

```
let MessageQueue = require('MessageQueue');

let BatchedBridge = new MessageQueue(
  __fbBatchedBridgeConfig.remoteModuleConfig,
  __fbBatchedBridgeConfig.localModulesConfig,
);

module.exports = BatchedBridge;
```

检查 MessageQueue 源码，其实是找不到 ToastAndroid 属性相关的声明的，因为这个属性是动态生成的。在分析 MessageQueue 源码之前，先大概看一下 MessageQueue 构造函数需要的两个参数。

```
__fbBatchedBridgeConfig.remoteModuleConfig,
__fbBatchedBridgeConfig.localModulesConfig,
```

__fbBatchedBridgeConfig 是一个全局 JS 变量，是在 CatalystInstance 中声明的，可以通过调用 ReactBridge.setGlobalVariable() 方法获得。setGlobalVariable 是在 Jni 中声明的方法，最终会调用 JavaScriptCore，从而将 Java 中定义的对象赋值给 JS 的全局对象 __fbBatchedBridgeConfig。__fbBatchedBridgeConfig 主要有两个属性：remoteModuleConfig 和 localModulesConfig。

其中，remoteModuleConfig 代表的是 Java 端中定义的模块，这些模块可以在 JS 中被调用。__fbBatchedBridgeConfig.remoteModuleConfig 的格式大概如下：

```
{
  "remoteModuleConfig": {
    "Logger": {
      "constants": { /* If we had exported constants... */ },
      "moduleID": 1,
      "methods": {
        "requestPermissions": {
          "type": "remote",
          "methodID": 1
        }
      }
    }
  }
}

{
  'ToastAndroid': {
    moduleId: 0,
    methods: {
      'show': {
        methodID: 0
      }
    },
    constants: {
```

```

    'SHORT': '0',
    'LONG': '1'
  },
  'moduleB': {
    moduleId: 0,
    methods: {
      'method1': {
        methodID: 0
      }
    },
    'key1': 'value1',
    'key2': 'value2'
  }
}

```

到此，再回头看一下 MessageQueue 的相关分析，在 MessageQueue 源码中有一个很重要的方法：`_genModules(remoteModules)`。

```

_genModules(remoteModules) {
  let moduleNames = Object.keys(remoteModules);
  for (var i = 0, l = moduleNames.length; i < l; i++) {
    let moduleName = moduleNames[i];
    let moduleConfig = remoteModules[moduleName];
    this.RemoteModules[moduleName] = this._genModule({}, moduleConfig);
  }
}

```

上面代码主要完成如下的事情：遍历 `remoteModules` 对象中所有的 key，得到 `moduleName`，然后针对每个 module 调用 `_genModule()` 方法生成 module 对象。

```

_genModule(module, moduleConfig) {
  let methodNames = Object.keys(moduleConfig.methods);
  for (var i = 0, l = methodNames.length; i < l; i++) {
    let methodName = methodNames[i];
    let methodConfig = moduleConfig.methods[methodName];
    module[methodName] = this._genMethod(
      moduleConfig.moduleID, methodConfig.methodID, methodConfig.type);
  }
  Object.assign(module, moduleConfig.constants);
  return module;
}

```

`_genModule()` 方法和 `_genModules()` 方法类似，遍历 module 下所有的方法，针对每个方法调用 `_genMethod()` 方法。

```

_genMethod(module, method, type) {
  ...
  fn = function(...args) {
    let lastArg = args.length > 0 ? args[args.length - 1] : null;

```

```

let secondLastArg = args.length > 1 ? args[args.length - 2] : null;
let hasSuccCB = typeof lastArg === 'function';
let hasErrorCB = typeof secondLastArg === 'function';
hasErrorCB && invariant(
  hasSuccCB,
  'Cannot have a non-function arg after a function arg.'
);
let numCBs = hasSuccCB + hasErrorCB;
let onSucc = hasSuccCB ? lastArg : null;
let onFail = hasErrorCB ? secondLastArg : null;
args = args.slice(0, args.length - numCBs);
return self.__nativeCall(module, method, args, onFail, onSucc);
};
}

```

`_genModule()` 方法最终会调用 `__nativeCall()` 方法。`__nativeCall()` 方法相关源码如下：

```

__nativeCall(module, method, params, onFail, onSucc) {
  if (onFail || onSucc) {
    // eventually delete old debug info
    (this._callbackID > (1 << 5)) &&
      (this._debugInfo[this._callbackID >> 5] = null);

    this._debugInfo[this._callbackID >> 1] = [module, method];
    onFail && params.push(this._callbackID);
    this._callbacks[this._callbackID++] = onFail;
    onSucc && params.push(this._callbackID);
    this._callbacks[this._callbackID++] = onSucc;
  }
  this._queue[MODULE_IDS].push(module);
  this._queue[METHOD_IDS].push(method);
  this._queue[PARAMS].push(params);

  var now = new Date().getTime();
  if (global.nativeFlushQueueImmediate &&
    now - this._lastFlush >= MIN_TIME_BETWEEN_FLUSHES_MS) {
    global.nativeFlushQueueImmediate(this._queue);
    this._queue = [[], [], []];
    this._lastFlush = now;
  }

  if (__DEV__ && SPY_MODE && isFinite(module)) {
    console.log('JS->N : ' + this._remoteModuleTable[module] + '.' +
      this._remoteMethodTable[module][method] + '(' + JSON.stringify(params) + ')');
  }
}

```

`__nativeCall()` 方法首先会检查是否有 `onFail` 和 `onSucc` 标志，如果有的话就直接压入 `_callbacks` 栈中，同时会将 `ModuleID`、`MethodID` 和 `Params` 等信息放入队列中。然后，通过调用 `nativeFlushQueueImmediate()` 方法将 C++ 代码注入到 JS 的全局变量中。

那 C++ 的代码又是如何注入到 JS 中的呢？答案就在 JSCExecutor.cpp 里面，通过调用 JSCExecutor.cpp 的 installGlobalFunction，installGlobalFunction 实际上是通过 JavaScriptCore 的 API 来实现 JS 调用 C++ 代码功能的。

```
installGlobalFunction(m_context, "nativeFlushQueueImmediate", nativeFlushQueueImmediate);
installGlobalFunction(m_context, "nativeLoggingHook", nativeLoggingHook);
installGlobalFunction(m_context, "nativePerformanceNow", nativePerformanceNow);
```

nativeFlushQueueImmediate 其实又调用了 flushQueueImmediate() 方法。

```
void JSCExecutor::flushQueueImmediate(std::string queueJSON) {
    m_flushImmediateCallback(queueJSON);
}
```

m_flushImmediateCallback 是在 JSCExecutor 的构造函数中完成初始化的，而 JSCExecutor 对象则是通过 JSCExecutorFactory 工厂对象的 createJSCExecutor() 方法创建的。

```
std::unique_ptr<JSCExecutor> JSCExecutorFactory::createJSCExecutor(FlushImmediateCallback cb) {
    return std::unique_ptr<JSCExecutor>(new JSCExecutor(cb));
}
```

m_flushImmediateCallback 函数作为实现 JS 层和 Native 层通信的核心，是通过 OnLoad.cpp 文件的 create 方法传入的，这个方法最终提供给 Java 层调用。这么看来，JS 层调用 Native API 的过程，最终就是调用 Bridge 传递过来的 ReactCallback 对象。

```
public ReactBridge(
    JavaScriptExecutor jsExecutor,
    ReactCallback callback,
    MessageQueueThread nativeModulesQueueThread) {
    .....
}
```

ReactBridge 对象最终是在 CatalystInstanceImpl 中创建的。

```
private ReactBridge initializeBridge() {
    bridge = new ReactBridge(
        jsExecutor,
        new NativeModulesReactCallback(),
        mCatalystQueueConfiguration.getNativeModulesQueueThread());
}

private class NativeModulesReactCallback implements ReactCallback {
    public void call(int moduleId, int methodId, ReadableNativeArray parameters) {
        mJavaRegistry.call(CatalystInstanceImpl.this, moduleId, methodId, parameters);
    }
}
```

ReactCallback 类的子类 NativeModulesReactCallback 最终调用了 call 方法，而 call 方法又调用了 mJavaRegistry.call 方法，mJavaRegistry 根据配置表查询到相应的方法完成调用。

7.1.2 React Native与iOS通信

在与 iOS 系统通信方面，React Native 使用 iOS 自带的 JavaScriptCore 作为 JS 的解析引擎，但它并没有直接使用 JavaScriptCore 提供的 JS 与 OC 互调特性，而是自己实现了一套机制。这套机制可以通用于所有 JS 引擎，在没有 JavaScriptCore 的情况下也可以用 Webview 代替。实际上项目里使用 Webview 作为引擎来替换 JavaScriptCore 引擎，也是为了兼容 iOS7 以下版本。

属性传递

跨组件通信最简单的例子就是属性的传递。例如，将原生组件传递属性到 React Native 中，使用 RCTRootView 将 React Native 视图封装到原生组件中。RCTRootView 是一个 UIView 容器，承载着 React Native 页面，其初始化方法里提供了一个接口，用于传递参数。

```
NSArray *imageList = @[@"http://foo.com/bar1.png",
                        @"http://foo.com/bar2.png"];
NSDictionary *props = @{@"images" : imageList};
RCTRootView *rootView = [[RCTRootView alloc] initWithBridge:bridge
                        moduleName:@"ImageBrowserApp"
                        initialProperties:props];
```

然后，在 JS 端直接调用这个接口即可。例如，下面是调用原生接口完成图片地址传递的代码。

```
renderImage: function(imgURI) {
  return (
    <Image source={{uri: imgURI}} />
  );
},
render() {
  return (
    <View>
      {this.props.images.map(this.renderImage)}
    </View>
  );
}
```

不过，由于信息在 React Native 中是单向流动的，所以，它的主要缺点是不支持回调，也就无法实现自下而上的数据绑定。设想，当前有一个小的 React Native 视图，当一个 JS 动作触发后，需要从原生的父视图中移除它。因为缺少回调机制，此时你会发现上面的需要根本做不到。

原生模块调用JS

在 React Native 中嵌入原生模块时，利用原生组件的 RCTViewManager 作为视图代理，通过 bridge 向 JS 发送调用事件。不过需要注意的是，发起一些不太常见的事件时，系统无法确保执行的时间，因为事件的处理函数是在单独的线程中执行。

那么，如果开发者自己创建的 JS 模块想要被系统模块调用，该怎么做呢？这时候只需要将 JS 模块注册添加到 messagequeue 的 _callableModules 中，然后在原生模块层调用相应的 JS 方法即可。

```
// 注册 JS 模块
BatchedBridge.registerCallableModule(
```

```

    'RCTEventEmitter',
    ReactNativeEventEmitter
  );
  // 原生调用 JS
  [_bridge enqueueJSCall:@("RCTEventEmitter.receiveEvent"
    args:body ? @[body[@"target"], name, body] : @
    [body[@"target"], name]);

```

JS调用原生模块

原生模块是 JS 中使用的 Objective-C 类 (OC)。一般来说, 这样的模块的每一个实例都是在每一次通过 JSbridge 通信时创建的, 它们可以导出任意的函数和常量给 React Native 使用。

JS 调用原生模块的完整流程如图 7-7 所示。

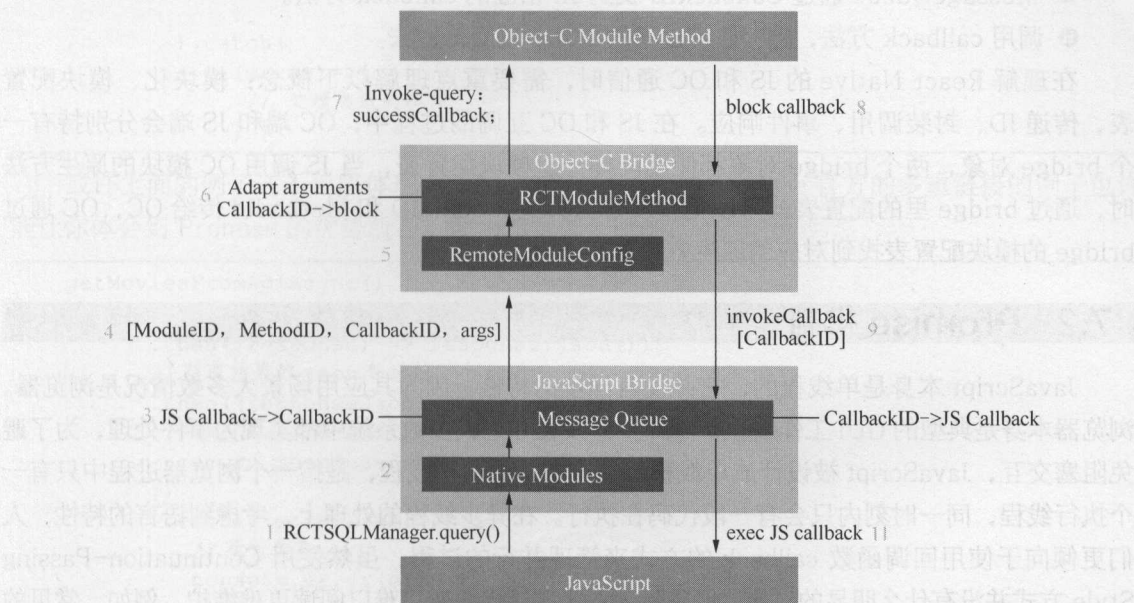


图7-7 JS调用OC流程图

为了详细说明 JS 发起调用到执行回调的整个流程, 下面对整个流程做精细的拆分, 主要涉及 11 个步骤。

- 1 JS 调用某个 OC 模块暴露出来的方法。
- 2 JS 将 JS 层的调用拆分为 ModuleID、MethodID 及 Params 分别推送到相应的 MessageQueue 中 (在初始化模块配置表的时候, 每一个模块都生成了对应的 remoteModule 对象, 对象里也生成了跟模块配置表里一一对应的方法)。
- 3 把 JS 的 callback 函数缓存在 MessageQueue 的成员变量里, 并把模块配置表里的 ModuleName 和 MethodName 转为 ModuleID 和 MethodID。
- 4 将返回的 ModuleID、MethodID、CallbackID 和其他参数传递给 OC。
- 5 OC 接收到消息, 通过模块配置表拿到对应的模块和方法。这里需要说明的是, 跟 JS 一

样,在初始化时,OC 也对模块配置表上的每一个模块生成了对应的实例并缓存起来,模块上的每个方法都会生成对应的 RCTModuleMethod 对象,通过 ModuleID 和 MethodID 取到对应的 RCTModuleMethod 实例和 Module 实例进行调用。

⑥ RCTModuleMethod 对 JS 传过来的每一个参数进行处理,处理完成之后,通过 NSInvocation 动态调用相应的 OC 模块方法。

⑦ OC 模块方法调用完后,执行 block 回调。

⑧ 调用 RCTModuleMethod 生成的 block 回调。

⑨ 使用 block 回调里的 CallbackID 和 block 传过来的参数去调用 JS 里的 MessageQueue invoke CallbackAndReturnFlushedQueue。

⑩ MessageQueue 通过 CallbackID 找到 JS 相应的 callback 方法。

⑪ 调用 callback 方法,把 OC 带过来的参数一起回传过去。

在理解 React Native 的 JS 和 OC 通信时,需要重点理解以下概念:模块化、模块配置表、传递 ID、封装调用、事件响应。在 JS 和 OC 互调的过程中,OC 端和 JS 端会分别持有一个 bridge 对象,两个 bridge 对象都使用同样一份模块配置表,当 JS 调用 OC 模块的原生方法时,通过 bridge 里的配置表就可以把模块方法转为 ModuleID 和 MethodId 传给 OC,OC 通过 bridge 的模块配置表找到对应的原生方法执行即可。

7.2 Promise 机制

JavaScript 本身是单线程的,它并没有异步的特性。因为其应用场景大多数情况是浏览器。浏览器本身是典型的 GUI 工作线程,GUI 工作线程在绝大多数系统中都实现为事件处理,为了避免阻塞交互,JavaScript 被设计成单线程工作方式。所谓单线程,是指一个浏览器进程中只有一个执行线程,同一时刻内只会有一段代码在执行。在异步线程的处理上,考虑到语言的特性,人们更倾向于使用回调函数 callback 的方式来管理并行的过程。虽然使用 Continuation-Passing Style 方式并没有什么明显的过错,但实际上,这样让代码变得难以阅读更难维护。例如,常见的多层回调函数的嵌套,即我们常说的“可怕金字塔”(Pyramid of Doom),是一种糟糕的编程方式。

在 Promise 异步机制出现之前,JavaScript 中最常见的做法就是通过回调函数来实现,即在回调内部再嵌套回调。例如:

```
asyncOperation(function(data){// 处理 'data'
  anotherAsync(function(data2){// 处理 'data2'
    yetAnotherAsync(function(){// 完成
    });
  });
});
```

7.2.1 Promise 简介

何为 Promise 异步机制呢? 通俗来讲,Promise 代表一个目前还不可用,但是在未来的

某个时间点可以被解析的值，它允许以一种同步的方式来编写异步代码。例如，使用 Promise API 执行异步调用远程服务，但是在发起请求前你并不知道返回的数据对象是什么样子。你可以创建一个 Promise 对象作为未来某个时间返回的数据对象，在此期间，Promise 对象扮演了真实数据的代理角色。

在 Promise 出现之前，异步操作往往是借助回调函数来实现的，而使用 Promise 机制后，只需要简单地调用 then 方法即可，这个接口专门用来处理 Promise 完成的成功或者失败的情况。其模块方法如下所示：

```
this.AsyncFunction(para).then(
  (para)=>{
    // 处理成功的事件
  }
).catch(
  (error)=>{
    // 处理失败的事件
  })
```

或许上面的例子还不足以体现 Promise 的强大，通过 Promise 官方的多重链接的例子也许能让你体会到 Promise 的优势所在。使用示例如下所示：

```
getMoviesFromApiAsync() {
  return fetch('http://facebook.github.io/react-native/movies.json')
    .then((response) => response.json())
    // 获取结果的 json 传递给下个 then
    .then((responseJson) => {
      // 执行成功获取结果
      return responseJson.movies;
    })
    .catch((error) => {
      // 执行失败
      console.error(error);
    });
}
```

Promise 将嵌套的 callback 改造成一系列 then 的连续调用，去除了层层嵌套的代码风格。它不仅有效地解决了回调金字塔问题，还提供了更好的代码组织模式。

Promise 对象

Promise 作为一个异步对象，在其创建时是未知的，当 Promise 对象状态发生改变后，它表示一个异步操作的最终结果，与之进行交互的方式主要是 then 方法，该方法注册了两个回调函数，分别是成功（resolve）和失败（reject）。其中，resolve 可以使 Promise 对象的状态改变为成功，同时传递一个参数用于成功后的操作。reject 则是将 Promise 对象的状态改变为失败，并将错误的信息用于错误处理操作上。在下面的例子中，传入一个回调函数初始化一个 Promise 对象，然后定义一个函数接受一个参数，如果传入 true，则打印“Hello World!”，否则打印错误信息。

```
function PromiseTest(ready) {
    return new Promise(function (resolve, reject) {
        if (ready) {
            resolve("Hello World!");
        }
    });
}
// 测试
PromiseTest(true).then(function (message) {
    console.log(message);
}, function (error) {
    console.log(error);
});
```

除了用上面的方式可以初始化 Promise 之外，ES6 还提供了 4 种常用的初始化方法，均返回一个 Promise 对象。

- `Promise.all(iterable)`: 返回一个新的 Promise 对象，当 `iterable` 中所有 Promise 对象都执行成功才会触发成功操作（`onFulfilled`），一旦有任何一个 `iterable` 里面的 Promise 对象执行失败，就立即退出并触发失败操作 `onRejected`。
- `Promise.race(iterable)`: 返回 Promise 对象，当 `iterable` 中有任意一个 Promise 被 `resolve` 后就立即退出，并触发 `onFulfilled`。
- `Promise.reject(reason)`: 返回用 Promise 包装的拒绝原因。调用 `then` 方法时，`onRejected` 将会接收到这个值。
- `Promise.resolve(value)`: 返回 Promise 对象用于值判断。如果 `value` 是 Promise 对象，则把 Promise 中 `resolve/reject` 的参数传递给对应的 `onFulfilled/onRejected`；如果 `value` 是 `thenable` 对象（有 `then` 属性的对象），就会把 `obj` 包装成 Promise 对象，该对象的 `then` 就是 `obj.then`；如果 `value` 是 `!thenable` 的值，会把该值包装成 Promise 对象，同时调用 `then` 方法时，`onFulfilled` 会接受这个值。

除此之外，Promise 对象拥有两个主要的方法，均返回一个 Promise 对象。

- `Promise.prototype.catch(onRejected)`: 返回一个 Promise 对象，如果触发 `onReject`，则返回对象的 `onReject` 的值；否则跳过 `catch`，直接返回原 Promise 对象。
- `Promise.prototype.catch(onRejected)`: 给 Promise 对象添加 `onFulfilled/onRejected` 回调 handler，当 Promise 的 `resolve/reject` 执行时触发对应的 handler。和 `promise.then` 不同的是，如果 Promise 已经处于 `fulfilled` 或 `rejected` 状态，那么相应的方法将会被立即调用。

Promise 状态

Promise 对象有 3 种状态：`pending`（等待）、`onFulfilled`（完成）、`onRejected`（拒绝）。Promise 语法如下：

```
new Promise(function(resolve, reject) {
```

```
...
});
```

其中, onFulfilled 和 onRejected 统称为 settled, pending 是初始状态。如果创建一个 Promise 后, func 中的 resolve 和 reject 都还没有执行, 此时 Promise 的状态为 pending。

对于 Promise 对象的 3 种状态, Promises A+ 规范中明确规定了: pending 可以转化为 onFulfilled 状态或 onRejected 状态且只能转化一次。需要说明的是, onFulfilled 状态和 onRejected 状态只能由 pending 转化而来, 两者之间不能互相转换, 如图 7-8 所示。

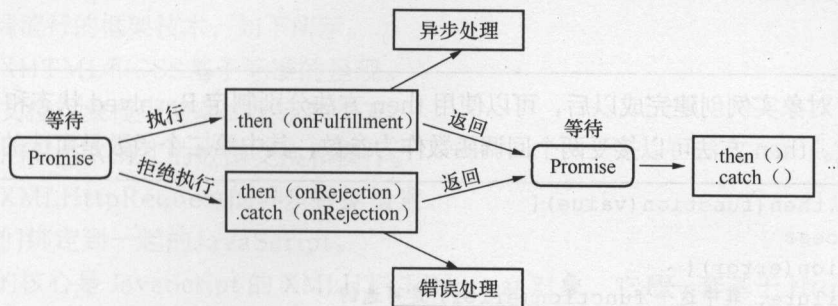


图7-8 Promise对象状态转换

当 then 被调用时, 构造器参数 resolve 和 reject 也将被调用。当执行 resolve 后, 触发 then 的 onFulfilled 回调, 并把 resolve 的参数传递给 onFulfilled; 当执行 reject 后, 触发 then/catch 的 onRejected 回调, 并把 reject 的参数传递给 onRejected。

注意: 如果 resolve 的参数是 Promise 对象, 且该 Promise 对象后面没有紧跟 then, 则该对象的 resolve/reject 会触发外层 Promise 对象 then 方法的 onFulfilled 或 onRejected。

then

一个 Promise 必须提供一个 then 方法以访问其当前值、终值和拒因。其中, then 方法接受两个参数:

```
promise.then(onFulfilled, onRejected)
```

onFulfilled 和 onRejected 都是可选参数。根据 Promises A+ 规范中的规定, onFulfilled 和 onRejected 在 Promise 执行相关函数前不可被调用, 并且其调用次数不能超过一次。

catch

catch 方法是 then 方法中 onRejected 函数的一个简单写法, 使用 catch 可以监听异常情况。即使你并不打算在代码中处理异常, 在代码中添加 catch, 也是严谨编程风格的体现。这意味着, 在使用 Promise 进行异步编程中, then 和 catch 总是成对出现的。

7.2.2 Promises基本用法

ES6 规定, Promise 对象是一个构造函数, 用来生成 Promise 实例。Promise 构造函数接

受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。我们可以通过 `new` 方法来创建 `Promise` 对象。

```
var promise = new Promise(function(resolve, reject){
  // ... 省 code
  // 异步操作结果，如果成功则执行 resolve，否则执行 reject
  if(成功){
    resolve(value);
  }else{
    reject(error);
  }
});
```

`Promise` 对象实例创建完成以后，可以使用 `then` 方法分别制定 `Resolved` 状态和 `Rejected` 状态的回调函数。`then` 方法可以接受两个回调函数作为参数，其中第二个函数是可选的。

```
promise.then(function(value){
  // success
},function(error){
  // failure, 其中这个 function(error) 是可选的
});
```

`Promise` 创建后就会立即执行：

```
promise.then(function(value){
  let promise = new Promise(function(resolve, reject){
    console.log('Promise');
    resolve();
  });

  promise.then(function(){
    console.log('Resolved');
  });

  console.log('Hi All');
  // 输出结果
  // Promise
  // Hi All
  // Resolved
```

上述代码中，首先输出的是“`Promise`”，然后 `then` 方法指定的回调函数，会在当前脚本所有任务执行完成之后执行，所以“`Resolved`”最后执行。

虽然，使用 `Promise` 可以很方便地控制异步操作。但是，`Promise` 机制也有一些缺点，例如新建 `Promise` 就会立即执行，而无法取消。其次，如果不设置回调函数，`Promise` 内部会报错，不会反应到外部。第三，当处于 `pending` 状态时，无法准确获知当前在哪个阶段。基于上面的问题，如果某些事件不断地反复发生，使用 `Stream` 模式比部署 `Promise` 更好。

7.2.3 在React Native中使用AJAX技术

AJAX 是 Asynchronous Javascript And XML (异步 JavaScript 和 XML) 的缩写, 是一种用于快速创建动态网页的技术。AJAX 适用于开发网站和应用程序, 通过在后台与服务器进行少量的数据交换, AJAX 就可以使网页实现异步更新。这意味着使用 AJAX 可以在不重新加载整个网页的情况下, 对网页的某部分进行局部更新。而使用传统的网页加载技术 (不使用 AJAX), 如果需要更新网页内容, 必须重载整个网页页面。

AJAX 不是一种编程语言, 而是一种用于创建更好更快以及交互性更强的动态网页技术, 它融合了前端流行的框架技术, 如下所示。

- 使用XHTML和CSS基于标准的呈现。
- 使用文档对象模型的动态显示和交互。
- 使用XML和XSLT的数据交换和操作。
- 使用XMLHttpRequest的异步数据检索。
- 将它们绑定到一起的JavaScript。

AJAX 的核心是 JavaScript 的 XMLHttpRequest 对象, 它是一套基于 Http 协议进行数据传送或接受 XML 及其他类型数据的 API, JavaScript 提供了一整套 XMLHttpRequest 对象 API, React Native 已经集成了这套 API。如果你想在移动应用中使用 AJAX 技术实现局部刷新技术, 也将是一个不错的选择。目前, W3C 已经制定了 AJAX 的相关标准和规范。

综合示例

下面使用 Promise 对象实现 AJAX 操作。相关代码如下:

```
var getJSON = function(url){
  var promise = new Promise(function(resolve, reject){
    var client = new XMLHttpRequest();
    client.open('GET', url);
    client.onreadystatechange = handler;
    client.responseType = 'json';
    client.setRequestHeader('Accept', 'application/json');
    client.send();

    function handler(){
      if(this.readyState !== 4){
        return;
      }
      if(this.status === 200){
        resolve(this.response);
      }else{
        reject(new Error(this.statusText));
      }
    }
  });
  return promise;
```



```

});

// 调用 getJSON 获取返回数据
getJSON('/posts.json').then(function(json) {
    console.log(json);
}, function(error) {
    console.log('出错了');
});

```

上述代码中，使用 Promise 创建一个 Promise 实例，然后在 Promise 的实例中创建 XMLHttpRequest 实例，并为 XMLHttpRequest 实例 client 设置状态监听，当 readyState 改变时，触发 onreadystatechange 事件并通知回调函数结果，然后通过调用 then 方法获取回调结果并刷新界面。

7.3 网络请求

在 AJAX 时代，都是通过 XMLHttpRequest 或者封装后的框架进行网络 API 请求的。而在前端技术快速发展的过程中，为了更好地契合一些优秀的设计模式，产生了 fetch 网络框架。相比 XMLHttpRequest，fetch 提供了更加强大高效的网络请求方式，并且 fetch 使用了 Promise 机制，这让它使用起来更加简洁。

7.3.1 XMLHttpRequest 请求

XMLHttpRequest 是一个 JavaScript 对象，它最初由微软设计，随后被 Mozilla、Apple 和 Google 等公司采纳，目前 AJAX 已经成为了 Web 应用的主流开发技术，并且由 W3C 组织标准化。

在 AJAX 时代，XMLHttpRequest 主要用来实现客户端和服务端之间的数据传输和数据交换。使用 XMLHttpRequest，可以很容易地取回一个 URL 上的资源数据。尽管名字里面包含 XML，但 XMLHttpRequest 可以取回所有类型的数据资源，并不局限于 XML。使用 XMLHttpRequest，你可以很轻松地实现网页的局部更新，而不会刷新整个页面。XMLHttpRequest 除了支持 HTTP 协议之外，还支持 FILE 和 FTP 协议。

构造函数

- XMLHttpRequest()

构造函数用于初始化一个 XMLHttpRequest 对象。

方法

XMLHttpRequest 提供的常用方法如下。

- abort()

如果请求已经发送，调用此方法可以终止请求。
- getAllResponseHeaders()

返回所有响应头信息（响应头名和值），如果响应头还没接受，则返回 null。
- getResponseHeader()

返回指定的响应头值，如果该响应头不存在，则返回 null。

- `open()`
初始化一个请求。该方法用于JavaScript代码中，如果是本地代码，使用 `openRequest()` 方法代替。参数如下。
- `method`: 请求所使用的HTTP方法，如GET、POST。
- `url`: 请求所要访问的URL。
- `async`: 是否执行异步操作。如果值为false，则`send()`方法不会返回任何东西，直到接收了从服务器返回的数据；如果为true，一个对开发者透明的通知会发送到相关的事件监听器统一。
- `user`: 用户名，可选参数，为授权使用。
- `password`: 密码，可选参数，为授权使用。
- `overrideMimeType()`
重写由服务器返回的 MIME type，例如，将响应流当作“text/xml”来处理。
该方法必须在 `send()` 之前被调用。
- `send()`
发送请求。如果该请求是异步模式（默认），该方法会立刻返回结果。如果请求是同步模式，则直到请求的响应完全接收以后，才会有返回结果。
- `setRequestHeader()`
给指定的HTTP请求头赋值。包含的参数如下。
- `header`: 被赋值的请求头名称。
- `value`: 被赋值的请求头值。

属性

- `readyState`
此属性返回一个XMLHttpRequest 代理当前所处的状态。状态值包括UNSENT、OPENED、HEADERS_RECEIVED、LOADING、DONE。
- `onreadystatechange`
此属性会在readyState发生改变时触发，当 readyState 的值改变的时候，callback 函数会被调用。
- `timeout`
请求在被自动终止前所消耗的时间。
- `withCredentials`
该属性表示是否使用类似cookies, authorization headers（头部授权）或者TLS客户端证书这一类资格证书，来创建一个跨站点访问控制请求。

XMLHttpRequest实例

使用XMLHttpRequest 来实现网络请求功能需要设置两个监听函数，分别用来处理成功和失败的情况，并且在发起网络请求的时候还需要依次调用 `open()` 和 `send()`。

```
function reqListener() {
    var data = JSON.parse(this.responseText); console.log(data);
}
function reqError(err) {
    console.log('Fetch Error : %S', err);
}
var oReq = new XMLHttpRequest();
oReq.onload = reqListener;
oReq.onerror = reqError; oReq.open('get', './api/some.json', true);
oReq.send();
```

例如，使用 XMLHttpRequest 发送一个 json 请求的代码如下：

```
var xhr = new XMLHttpRequest();
xhr.open('GET', url);
xhr.responseType = 'json';

xhr.onload
    = function() {
        console.log(xhr.response);
    };

xhr.onerror = function() {
    console.log("Oops, error");
};
xhr.send();
```

XMLHttpRequest 作为 AJAX 技术的核心，最大的优点是在不重新加载页面的情况下实现网页的更新。不过对于日益复杂的网络请求，XMLHttpRequest 看起来显得越来越糙。

7.3.2 fetch 请求

基于 XMLHttpRequest 的 AJAX 技术是一种有效改进页面通信的技术。不过随着 fetch 技术的出现，XMLHttpRequest 看起来显得越来越笨拙和粗糙。传统的基于 XMLHttpRequest 技术不符合分离原则，配置和调用非常混乱，而基于事件的异步模型也没有采用 Promise 机制的 async/await 方式友好。fetch 的出现正是为解决 XHR 的问题而生。

例如，使用 fetch 发送 json 请求，看起来就非常简洁。

```
fetch(url).then(response => response.json())
    .then(data => console.log(data))
    .catch(e => console.log("Oops, error", e))
```

fetch 语法

使用 fetch 进行网络请求则更加简单，只需简单地将网址作为参数传递给 fetch 的构造方法即可。

```
fetch('./api/some.json')
    .then(function(res) {
        if (res.status !== 200) {
```

```

        console.log(' Status Code: ' + res.status); return;
    }
    // 处理响应数据
  });
}
).catch(function(err) {
  console.log('Fetch Error : %S', err);
})

```

fetch请求

fetch 不仅支持 GET、POST 网络请求方式，还支持自定义 Headers 对象和请求参数。不过，虽然 fetch 相比 XMLHttpRequest 是进步了不少，不过也有一些不足之处。Promises 缺少了一些重要的 XMLHttpRequest 的使用场景。例如，使用标准的 ES6 Promise 的时候无法收集进入信息或中断请求。

下面简单介绍一下使用 fetch 进行网络请求的几种方式。

使用 GET 方式进行网络请求：

```

fetch('./api/some.json ', {
  method: 'GET'
}).then(function(response) {
  // 获取数据，数据处理
}).catch(function(err) {
  // 错误处理
});

```

使用 POST 方式进行网络请求：

```

var url = './api/some.json';
let param = {user: 'xxx', phone: 'xxxxxxx'};
fetch(url, {
  method: 'post',
  body: JSON.stringify(param)
}).then(function(response) {
  // 获取数据，数据处理
});

```

React Native 除了支持常用的 GET、POST 方式请求之外，还支持添加消息头信息操作。在使用 HTTP 协议进行网络请求时，React Native 系统默认支持 gzip/deflate 格式编码，开发者不需要进行额外设置。例如：

```

var url = './api/some.json';
fetch(url, {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({

```



```

    })
  })

```

WebSocket

React Native 除了可以支持 POST、GET 请求之外，还支持 WebSocket 全双工的通信信道。

```

var ws = new WebSocket('ws://host.com/path');
ws.onopen = () => {
  // 打开一个连接
  ws.send('something'); // 发送一个消息
};
ws.onmessage = (e) => {
  // 接收到了一个消息
  console.log(e.data);
};
ws.onerror = (e) => {
  // 发生了一个错误
  console.log(e.message);
};
ws.onclose = (e) => {
  // 连接被关闭了
  console.log(e.code, e.reason);
};

```

在 React Native 中，使用 fetch 进行网络请求，可能会碰到如下问题。

请求时出现异常：如果报 server 端参数格式错误，将 header 里面的 Content-Type 设置为 application/x-www-form-urlencoded 即可。

响应时出现异常：响应时出现异常通常是 response.json() 这一句出现异常。可以根据实际情况做相应的处理。

超时问题：目前，fetch 本身没有提供设置超时的属性，只能通过 Promise 机制来进行超时设置。

fetch网络请求示例

需要注意的是，iOS 9 之后，系统强制要求使用 HTTPS 进行网络请求，否则会提示错误。如图 7-9 所示是使用 fetch 方式请求数据，并将结果绘制成九宫格界面的例子。

具体的在组件的 componentDidMount 函数中发出 fetch 请求，当服务端返回 json 数据之后，再通过 React Native 的状态机机制通知数据源发生变化，进而通知界面重新刷新。示例代码如下：

```

var url='';// 请求地址
class RecommendScene extends Component {
  state: {
    discounts: Array<Object>,

```

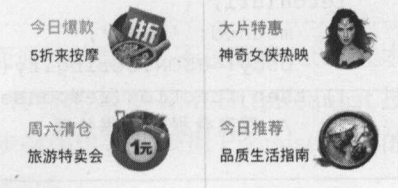


图7-9 fetch网络请求

```

    dataSource: ListView.DataSource
  }
  constructor(props: Object) {
    super(props)
    let ds = new ListView.DataSource({ rowHasChanged: (r1, r2) => r1 !== r2 })
    this.state = {
      discounts: [],
      dataSource: ds.cloneWithRows([]),
    }
  }

  componentDidMount() {
    this.requestDiscount();
  }

  onGridSelected(index: number) {
    alert(index)
  }

  // 请求打折产品
  requestDiscount() {
    fetch(url)
      .then((response) => response.json())
      .then((json) => {
        console.log(JSON.stringify(json));
        this.setState({ discounts: json.data })
      })
      .catch((error) => {
        alert(error)
      })
  }

  render() {
    return (
      <View style={styles.container}>
        // 绘制界面
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#f3f3f3',
    marginTop: 20
  },
});

export default RecommendScene;

```

在实际项目开发中，为了对项目的网络请求进行统一的管理，需要对 fetch 请求进行二次

封装。封装过程中，需要将一些公共的部分包装起来，然后将不同的部分通过参数的方式暴露出来。示例代码如下：

```
var HTTPUtil = {};

// GET 请求
HTTPUtil.get = function(url, params, headers) {
  if (params) {
    let paramsArray = [];
    //encodeURIComponent
    Object.keys(params).forEach(key => paramsArray.push(key + '=' + params[key]))
    if (url.search(/\?/) === -1) {
      url += '?' + paramsArray.join('&')
    } else {
      url += '&' + paramsArray.join('&')
    }
  }
  return new Promise(function (resolve, reject) {
    fetch(url, {
      method: 'GET',
      headers: headers,
    })
      .then((response) => {
        if (response.ok) {
          return response.json();
        } else {
          reject({status: response.status})
        }
      })
      .then((response) => {
        resolve(response);
      })
      .catch((err) => {
        reject({status: -1});
      })
  })
}

// POST 请求 FormData 表单数据
HTTPUtil.post = function(url, formData, headers) {
  return new Promise(function (resolve, reject) {
    fetch(url, {
      method: 'POST',
      headers: headers,
      body: formData,
    })
      .then((response) => {
        if (response.ok) {
          return response.json();
        }
      })
  })
}
```

```

    } else {
      reject({status:response.status})
    }
  })
  .then((response) => {
    resolve(response);
  })
  .catch((err)=> {
    reject({status:-1});
  })
})
}
export default HTTPUtil;

```

7.4 小结

网络通信，是软件开发中的核心技术，它将各自孤立的设备连接起来，从而实现资源共享。在项目中，合理地选取请求方式和处理开发中的各种网络情况，合理地处理网络开发中各种情况，有助于我们开发高质量的移动应用。而通过对本地通信技术的相关分析，可以让我们深层次地理解 React Native 的设计原理。

React Native 和原生系统之间的通信，让我们对 React Native 采用的跨平台方案有了一个全新的认识，这为混合开发提供技术参考；采用异步 Promise 编程是前端开发的基本技术，并获取了越来越多的浏览器厂商的支持。基于 fetch 的网络请求是对 XMLHttpRequest 技术的改进方案采用 fetch 方式，提高了代码的可阅读性和可维护性。

第

8

章

混合开发高级篇

有时候你想要访问平台组件或者 API，但 React Native 可能还没有相应的模块供开发者使用；或者你需要复用 Objective-C、Swift 或 Java 代码，而不是使用 JavaScript 重新实现一遍；又或者你需要拓展某些已有的组件或 API 以实现特定的需求。种种情况说明，需要 React Native 具备平台互访的能力。

React Native 在设计之初就考虑到跨平台调用的问题，并为开发者提供了访问平台的能力。这是一个相对高级的特性，在日常开发的过程中并不会经常出现，但具备这样的能力是很重要的。如果 React Native 还不支持某个你需要的原生特性，你也可以自己实现该特性的封装。

在如今的移动应用程序中，已经有成千上万的原生 UI 组件，其中的一些是平台的，另一些可能来自于第三方库，甚至你也可以实现组件。React Native 已经封装了大部分最常见的组件，但不可能封装全部的原生组件。不过庆幸的是，React Native 封装和植入原生组件非常简单。

8.1 React Native调用iOS原生组件

在 React Native 混合开发中，系统允许开发者直接使用原生代码实现的 UI 模块或者其他自定义的 React Native 组件，它们称为私有 React Native 组件，封装好的私有组件可以被直接调用。不过，混合开发需要开发者具备一定的 Android/iOS 原生编程能力。

使用 Objective-C 或者 Swift 封装 React Native 组件的相关技术，React Native 官方文档已经写得很详细了，开发者只需要按照步骤去实现即可，官方文档地址：<http://facebook.github.io/react-native/docs/native-components-ios.html>，也可以访问中文地址：<http://>

reactnative.cn/docs/0.46/native-component-ios.html#content。

8.1.1 React Native链接原生库

要实现 ReactNative 调用 iOS 原生组件的目的,可以将互不干扰的功能模块打包成静态库。细心的读者可能会发现,在创建 peact Nctive 项目的时候,一些库已经布置到 Libraries 文件夹下面。其中有一些是纯 JavaScript 代码实现,只需要使用 require 的方式导入即可使用。另外有一些库是基于原生代码实现,使用前,必须把这些库添加到你的应用中,否则应用会在你使用这些库的时候报错。添加原生库需要几个步骤。

❶ 安装一个带原生依赖的库,命令如下:

```
npm install 某个带有原生依赖的库 --save
```

说明:--save 或 --save-dev 是包依赖的两种方式,具体使用的时候。React Native 会根据 package.json 文件中的 dependencies 和 devDependencies 记录来链接相应的库。

❷ 链接原生库依赖,命令如下:

```
react-native link
```

❸ 如果该库包含原生代码,还需要进行手动链接。在库的文件夹下有一个 .xcodproj 文件,将文件拖到你的 Xcode 工程下(通常拖到 Xcode 的 Libraries 分组目录下),如图 8-1 所示。

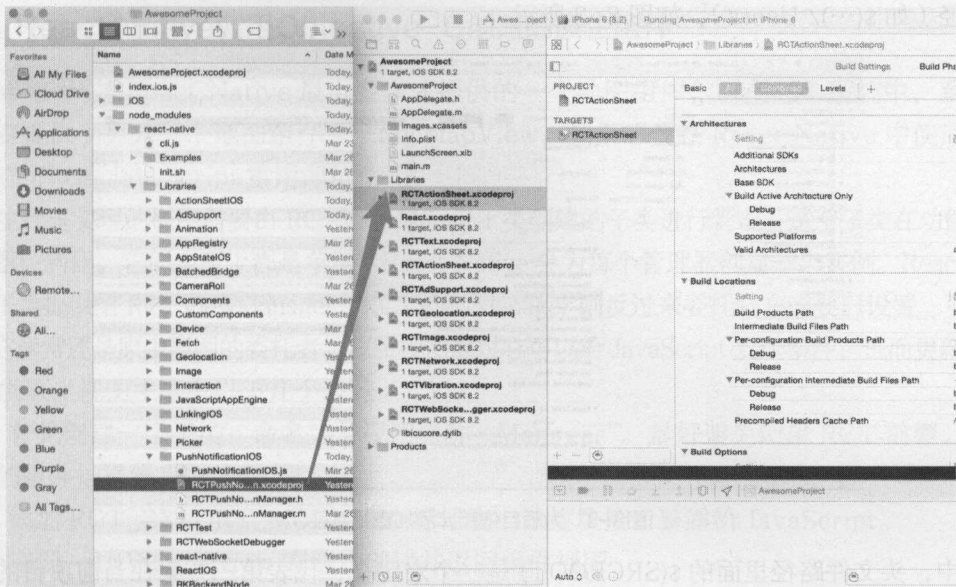


图8-1 添加项目库依赖

❹ React Native 不仅支持系统库,还支持一切用原生代码写成的第三方库。导入的方式和导入系统库类似,在项目上依次选择【Targets】→【Build Phases】→【Link Binary With Libraries】添加第三方类库生成的静态链接库即可,如图 8-2 所示。

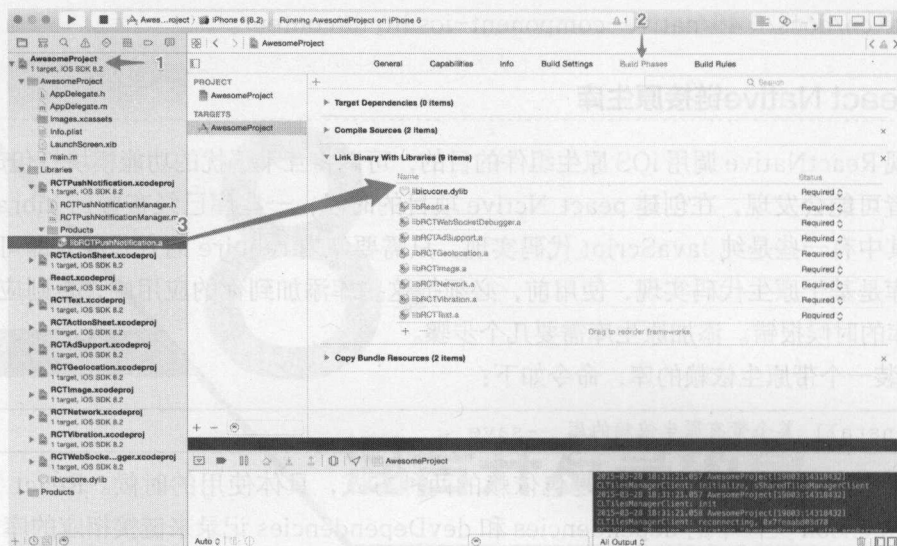


图8-2 链接项目库

⑤ 项目库添加后，有的库还需要为项目添加查找路径。读者可以按照下面的次序添加，选择【Targets】→【Build Settings】→【Search Paths】→【User Header Search Paths】，在参数中加入第三方类库的头文件路径即可，可以是绝对路径（如 /Users/libpath），也可以是相对路径（如 \$(…)/Users），如图 8-3 所示。

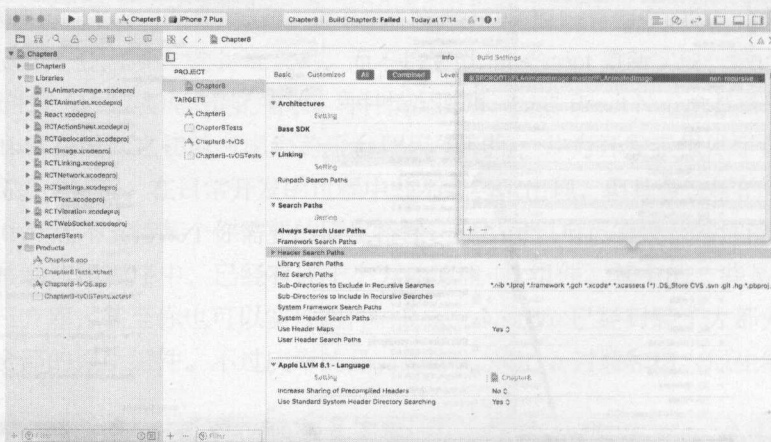


图8-3 为项目依赖库添加查找路径

其中，头文件路径里面的 \$(SRCROOT) 是一个宏定义，代表第三方案在当前项目的绝对路径，只需要在后面添加第三方库头文件在项目中的路径即可。如本例的路径为 \$(SRCROOT)/FLAnimatedImage-master/FLAnimatedImage。

项目默认不支持导入 React Native，需要手动导入。在弹出的窗口中点击加号，添加如下内容：\$(SRCROOT)/../node_modules/react-native/React，如图 8-4 所示。

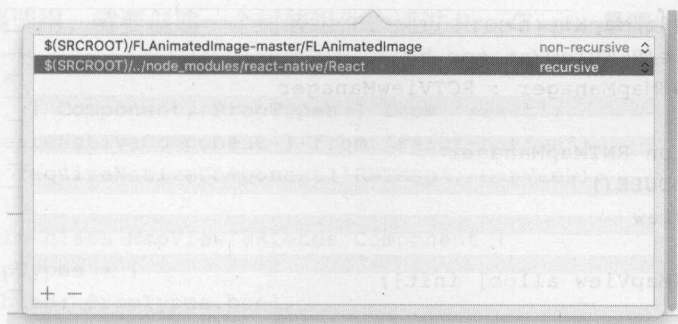


图8-4 添加头文件

在链接原生库的过程中，有以下几点需要注意。

- 如果第三方库只有.h和.a文件，在项目中直接添加.a文件即可。
- 如果第三方类库封装了一些资源在.bundle文件里，那么需要将.bundle文件和.xcodeproj一起拖到项目中。
- 有些静态链接库引用进来可能还需要增加一些标记，可以在【Targets】→【Build Settings】→【Linking】的【Other Linker Flags】参数中增加标记。
- 第三方类库引用了一些系统自带类库，如果项目中没有引用，可能会引起编译错误。所以，对于引用了系统库的第三方库，记得在使用的时候导入系统类库。

8.1.2 React Native调用Objective-C创建的原生组件

MapView 是 React Native 最近版本才提供的一个地图组件，而在 iOS 系统中，系统早就为开发者提供了另外一个地图组件——MKMapView。如果想要在 React Native 中使用它，只需要让它可以被 JavaScript 重用即可。

原生代码实现的视图需要由 RCTViewManager 类创建的子类进行管理。这个子类在功能上有些类似视图控制器，它们本质上都是单例，React Native 会为每个管理器创建一个实例。React Native 将这个子类提供给 RCTViewManager，RCTViewManager 则反过来委托它在需要时设置、更新视图属性，它通常还是原生视图的委托者，原生视图通过它可以给 JavaScript 发送事件，进而更新界面。

提供原生视图主要有以下几个步骤。

- ① 首先创建一个子类：命名为“视图名称 + Manager”。最好避免使用 RCT 前缀，除非你想把你的组件推送给官方使用。
- ② 添加 RCT_EXPORT_MODULE() 标记宏：让模块接口暴露给 JavaScript。
- ③ 实现 -(UIView *)view 方法：创建并返回组件视图。
- ④ 封装属性与事件传递。

现在，我们要在项目中建立一个 RCTViewManager 类的子类——RNTMap Manager.m——来管理原生视图。相关代码如下。

```
// RNTMapManager.m
```



```
#import <MapKit/MapKit.h>
#import <React/RCTViewManager.h>
@interface RNTMapManager : RCTViewManager
@end
@implementation RNTMapManager
RCT_EXPORT_MODULE()
- (UIView *)view
{
    return [[MKMapView alloc] init];
}
@end
```

接下来，需要编写 JavaScript 代码来让这个视图变成一个可用的 React 组件 MapView.js，然后使用 module exports 导出组件。

```
// MapView.js
var { requireNativeComponent } = require('react-native');
// requireNativeComponent 自动把这个组件提供给 "RNTMapManager"
module.exports = requireNativeComponent('RNTMap', null);
```

现在已经实现了一个简单的地图组件，但是还不能在 JavaScript 端真正地调用它。为此，我们还需要封装一些原生属性供 JavaScript 使用。例如，禁止手指捏放操作，那么可以在 RNTMapManage 文件里定义一个布尔值类型的属性。

```
// RNTMapManager.m
RCT_EXPORT_VIEW_PROPERTY(pitchEnabled, BOOL)
```

React Native 使用 RCTConvert 来完成 JavaScript 和原生代码之间的类型转换，如果转换无法完成，会产生红屏提示。JavaScript 与标准数据类型对应的关系如表 8-1 所示。

表 8-1 标准类型与 JS 类型类比

标准类型	JavaScript 类型
Boolean	Bool
Integer、Double、Float	Number
String	String
Callback	Function
ReadableMap	Object
ReadableArray	Array

要想在 MapView 组件里禁用捏放操作，还需要在组件使用的地方设置相应的属性。对应的代码如下：

```
// MyApp.js
<MapView pitchEnabled={false} />
```

为了方便 JS 端调用，需要创建一个封装组件，通过 PropTypes 暴露组件的接口。

```
// MapView.js
import React, { Component, PropTypes } from 'react';
import { requireNativeComponent } from 'react-native';
var RNTMap = requireNativeComponent('RNTMap', MapView);

export default class MapView extends Component {
  static propTypes = {
    pitchEnabled: PropTypes.bool,
  };
  render() {
    return <RNTMap {...this.props} />;
  }
}
```

到此，一个完整的 React Native 调用 objective 创建的原生组件的流程就走完了，接下来添加一个更复杂的原生属性，然后看一下实现流程。

```
// RNTMapManager.m
RCT_CUSTOM_VIEW_PROPERTY(region, MKCoordinateRegion, RNTMap)
{
  [view setRegion:json ? [RCTConvert MKCoordinateRegion:json] : defaultView.
region animated:YES];
}
```

现在 RNTMapManger 文件多了一个需要做类型转换的 MKCoordinateRegion 类型，还添加了一部分自定义的代码，该部分代码的作用是在 JS 里改变地图的可视区域的时候，视角会平滑地移动过去。在我们提供的函数体内，json 代表了 JS 中传递的尚未解析的原始值。另一个参数 defaultView 代表可以访问视图实例。

下面是 MKCoordinateRegion 转换函数的具体实现，它通过两个 RCTConvert 的扩展来完成。

```
@implementation RCTConvert(CoreLocation)

RCT_CONVERTER(CLLocationDegrees, CLLocationDegrees, doubleValue);
RCT_CONVERTER(CLLocationDistance, CLLocationDistance, doubleValue);

+ (CLLocationCoordinate2D)CLLocationCoordinate2D:(id)json
{
  json = [self NSDictionary:json];
  return (CLLocationCoordinate2D){
    [self CLLocationDegrees:json[@"latitude"]],
    [self CLLocationDegrees:json[@"longitude"]]
  };
}

@end
```

```

@implementation RCTConvert (MapKit)

+ (MKCoordinateSpan)MKCoordinateSpan:(id)json
{
    json = [self NSDictionary:json];
    return (MKCoordinateSpan){
        [self CLLocationDegrees:json[@"latitudeDelta"]],
        [self CLLocationDegrees:json[@"longitudeDelta"]]
    };
}

+ (MKCoordinateRegion)MKCoordinateRegion:(id)json
{
    return (MKCoordinateRegion){
        [self CLLocationCoordinate2D:json],
        [self MKCoordinateSpan:json]
    };
}

```

为了方便 JavaScript 调用原生属性还需要在 MapView.js 里添加相应的属性说明，属性和调用示例如下：

```

MapView.propTypes = {
    pitchEnabled: React.PropTypes.bool,
    region: React.PropTypes.shape({
        latitude: React.PropTypes.number.isRequired,
        longitude: React.PropTypes.number.isRequired,
        // 最小 / 最大经、纬度间的距离
        latitudeDelta: React.PropTypes.number.isRequired,
        longitudeDelta: React.PropTypes.number.isRequired,
    }),
};

// MyApp.js 调用
render() {
    var region = {
        latitude: 37.48,
        longitude: -122.16,
        latitudeDelta: 0.1,
        longitudeDelta: 0.1,
    };
    return <MapView region={region} />;
}

```

现在，已经完成原生地图组件的封装，通过新增的属性，JS 可以很容易地控制视图的滑动。不过我们怎么处理来自用户的事件呢？关键的步骤是在 RNTMapManager 中声明一个事

件处理函数（如命名为 `onChange()`）。此函数专门用来处理将原生事件传递给 JavaScript。

```
// RNTMap.h

#import <MapKit/MapKit.h>
#import <React/RCTComponent.h>
@interface RNTMap: MKMapView
@property (nonatomic, copy) RCTBubblingEventBlock onChange;
@end

实现类 RNTMap.m
```

```
// RNTMap.m

#import "RNTMap.h"
@implementation RNTMap
@end
```

接下来，需要将原生视图代码交由 `RCTViewManager` 创建的子类进行管理。通过继承，`MKMapView` 添加事件处理函数，然后将 `onChange` 方法暴露出来，委托 `RNTMapManager` 代理创建相关视图。

```
#import "RNTMapManager.h"
#import <MapKit/MapKit.h>
#import "RNTMap.h"
#import <React/UIView+React.h>

@interface RNTMapManager() <MKMapViewDelegate>
@end

@implementation RNTMapManager
RCT_EXPORT_MODULE()

RCT_EXPORT_VIEW_PROPERTY(onChange, RCTBubblingEventBlock)

- (UIView *)view
{
    RNTMap *map = [RNTMap new];
    map.delegate = self;
    return map;
}

#pragma mark MKMapViewDelegate

- (void)mapView:(RNTMap *)mapView regionDidChangeAnimated:(BOOL)animated
{
    if (!mapView.onChange) {
        return;
    }
}
```

```

MKCoordinateRegion region = mapView.region;
mapView.onChange(@{
    @"region": @{
        @"latitude": @(region.center.latitude),
        @"longitude": @(region.center.longitude),
        @"latitudeDelta": @(region.span.latitudeDelta),
        @"longitudeDelta": @(region.span.longitudeDelta),
    }
});
}

```

在上述代码的委托方法 `regionDidChangeAnimated` 方法中，根据对应的视图调用事件处理函数并传递区域数据，然后，`onChange` 事件会触发 JavaScript 端的同名回调函数。

```

class MapView extends React.Component {
    static propTypes = {
        /**
         * Callback that is called continuously when the user is dragging the map.
         */
        onChange: React.PropTypes.func,
        ...
    };
    _onChange = (event: Event) => {
        if (!this.props.onRegionChange) {
            return;
        }
        this.props.onRegionChange(event.nativeEvent);
    }
    render() {
        return <RCTMap {...this.props} onChange={this._onChange} />;
    }
}

```

```

class MapViewExample extends React.Component {
    onRegionChange(event: Event) {
        // Do stuff with event.region.latitude, etc.
    }

    render() {
        var region = {
            latitude: 37.48,
            longitude: -122.16,
            latitudeDelta: 0.1,
            longitudeDelta: 0.1,
        };

        return (

```

```

    <MapView region={region} pitchEnabled={false} style={{flex: 1}} onChange= {this.
onRegionChange}/>
    );
  }
}

// Module name
+AppRegistry.registerComponent('MapViewExample', () => MapViewExample);

```

React Native 为方便开发者封装原生视图组件，特意为其原生代码视图类提供了相应的生命周期函数。

- `insertReactSubview()`: 当原生视图被要求加入子视图时，此函数被调用。
- `removeReactSubview()`: 当原生视图的某个子视图被移除时，此函数被调用。
- `layoutSubview()`: 当原生视图将被渲染时，此函数被调用。它对应 React Native 的 `componentWillMount` 事件。
- `removeFromSuperview()`: 当原生视图将被渲染时，此函数被调用。它对应 React Native 的 `componentWillUnmount` 事件。

8.2 React Native调用Android原生组件

到目前为止，React Native 并没有为开发者提供视频播放相关的组件。如果要播放视频的话，有两种方法：一种是借助 `WebView` 组件来播放视频，一种就是通过系统提供的跨平台技术调用原生的播放器组件。第一种方式比较简单，此外不做讲解，本节主要讲解借助跨平台技术调用 Android 原生组件放视频的方法。

React Native 调用 Android 原生组件和 API 主要有两个大的步骤：编写原生 UI 组件和编写 React Native 端的调用代码。

8.2.1 编写原生UI组件

使用 Webstrom 创建一个 React Native 项目，然后使用 Android Studio 打开 Android 项目。如果开发者想实现 Android 系统没有提供的自定义原生组件，那么这个组件必须是 Android SDK 的 `View` 或者 `ViewGroup` 的子类。

本例中，`VideoView` 作为 Android 原生的视频播放组件，这是 React Native 目前还没有提供的组件。如果要在 React Native 中调用原生视频播放组件，在原生 Android 端需要完成如下的改造。

❶ 创建 `ViewManager` 子类，实现原生 UI 管理。

新建 `ViewManager` 的子类 `VideoViewManager`，并继承 `SimpleViewManager`，`SimpleViewManager` 需要传入一个泛型，该泛型为继承 Android 的 `View`。在本例中即 `VideoView`。相关代码如下。

```

public class VideoViewManager extends SimpleViewManager<VideoView>{
    @Override
    public String getName() { // 组件名称
        return "VideoView";
    }
    @Override
    protected VideoView createViewInstance(ThemedReactContext reactContext) {
        VideoView video = new VideoView(reactContext);
        return video;
    }
}

```

其中，getName() 返回组件名称，createViewInstance() 返回实例对象，可以在这个函数初始化对象时设置一些属性。

在原生视频开发中，调用 VideoView 组件播放视频的时候需要一个视频文件路径，然后调用 start() 方法即可实现视频的播放。相关功能代码如下：

```

@ReactProp(name = "source")
public void setSource(RCTVideoView videoView, @Nullable String source){
    if(source != null){
        videoView.setVideoURI(Uri.parse(source));
        videoView.start();
    }
}

```

在实际项目中，为了增强用户体验，并且减少代码的复杂性，需要对视频播放部分代码做一个简单的封装处理。例如，下面是通过简单封装后的视频播放代码：

```

@ReactProp(name = "source")
public void setSource(VideoView videoView, @Nullable ReadableMap source){
    if(source != null){
        if (source.hasKey("url")) {
            String url = source.getString("url");
            FLog.e(VideoViewManager.class, "url = "+url);
            HashMap<String, String> headerMap = new HashMap<>();
            if (source.hasKey("headers")) {
                ReadableMap headers = source.getMap("headers");
                ReadableMapKeySetIterator iter = headers.keySetIterator();
                while (iter.hasNextKey()) {
                    String key = iter.nextKey();
                    String value = headers.getString(key);
                    FLog.e(VideoViewManager.class, key+" = "+value);
                    headerMap.put(key, value);
                }
            }
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
                videoView.setVideoURI(Uri.parse(url), headerMap);
            }
        }
    }
}

```

```

    }else{
        try {
            Method setVideoURIMethod = videoView.getClass().getMethod(
("setVideoURI", Uri.class, Map.class);
            setVideoURIMethod.invoke(videoView, Uri.parse(url), headerMap);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    videoView.start();
}
}
}

```

React Native 通过 `@ReactProp` (或者 `@ReactPropGroup`) 注解来导出属性的设置方法, 该方法有两个参数: 第一个参数是泛型 `View` 的实例对象; 第二个参数是要设置的属性值, 方法的返回值类型必须为 `void`, 而且访问控制必须声明为 `public`。React Native 属性类型由函数第二个参数的类型决定, 支持的类型有: `boolean`、`int`、`float`、`double`、`String`、`Boolean`、`Integer`、`ReadableArray` 和 `ReadableMap`。

`@ReactProp` 注解必须包含一个字符串类型的参数 `name`。这个参数指定了对应属性在 JavaScript 端的名字。例如在本例中, 参数 `name` 为 `source`, 在 JavaScript 端调用的对应代码如下:

```
<VideoView source='http://ohe65w0xx.bkt.clouddn.com/test3.mp4'/>
```

❷ 创建原生 UI 管理类, 并向系统注册原生 UI 管理类。

接着, 开发者需要建立一个原生 UI 管理类, 在 Android 项目文件中新建一个 `VideoViewPackage` 类。代码如下:

VideoViewPackage.java

```

public class VideoViewPackage implements ReactPackage {
    @Override
    public List<NativeModule> createNativeModules(ReactApplicationContext
reactContext) {
        return Collections.emptyList();
    }

    @Override
    public List<Class<? extends JavaScriptModule>> createJSModules() {
        return Collections.emptyList();
    }

    @Override
    public List<ViewManager> createViewManagers(ReactApplicationContext
reactContext) {
        return Arrays.<ViewManager>asList(
            new VideoViewManager()

```



```

    );
  }
}

```

然后向系统注册这个原生 UI 管理类，以前的版本是在 MainActivity 中添加，最近的版本是改在 MainApplication 的 getPackages() 中注册。相关代码如下：

```

@Override
protected List<ReactPackage> getPackages() {
    return Arrays.<ReactPackage>asList(
        new MainReactPackage(),
        // 向系统注册自己实现的组件
        new OrientationPackage(),
        new VideoViewPackage()
    );
}

```

至此，原生部分的改造就基本完成了，接下来需要在 JS 端实现对原生部分的调用。

8.2.2 编写JavaScript端实现

❶ 建立私有的 JavaScript 文件，实现 React Native 端组件。

使用 WebStorm 打开 React Native 项目，在项目根目录下新建一个文件夹存放 JS 文件，然后新建一个 VideoPlayView.js 文件，相关代码如下：

VideoPlayView.js

```

import React, { PropTypes, Component } from 'react';
import {
    requireNativeComponent, View,
} from 'react-native';

var VideoView = {
    name: 'VideoView',
    propTypes: {
        style: View.propTypes.style,
        source: React.PropTypes.shape({
            url: React.PropTypes.string,
            headers: React.PropTypes.object,
        }),
        ...View.propTypes,
    }
};

var RCTVideoView = requireNativeComponent('VideoView', VideoView);
module.exports = RCTVideoView;

```

和导入自定义模块的 Modules 不同，组件使用的是 requireNativeComponent，requireNative

Component 接受两个参数：第一个参数是原生视图的名字（Java 层 VideoViewManager\$getName 的值）；而第二个参数是一个描述组件接口的对象。然后需要给组件声明一些属性如 name、propTypes 等。需要注意的是，View.propTypes 包含了默认 View 的属性。

② 在项目中引入 JavaScript 文件，实现私有调用。VideoPlayView.js 相关代码如下：

```
export default class VideoPlayView extends Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <View style={styles.videoContainer}>

        <Text style={styles.text}> 康熙王朝 </Text>
        <VideoView
          style={styles.video}
          source={
            {
              url: 'http://ohe65w0xx.bkt.clouddn.com/test3.mp4',
              headers: {
                'refer': 'myRefer'
              }
            }
          }
        />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  videoContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  text: {
    fontSize: 20,
    justifyContent: 'center',
  },
  video: {
    marginTop: 10,
    height: 250,
    width: 380
  },
});
```

运行效果如图 8-5 所示。



图8-5 运行效果

虽然，通过上面的简单封装，基本实现了调用原生组件播放视频的功能，但是，对于更加复杂的视频操作还无法实现，例如暂停视频、获取视频的总长度、当前视频的位置等，而原生的组件提供了相应的方法。

所以，要实现对视频的更多操作，实现对 VideoView 状态和视频信息的监听，就需要在 VideoView 的子类 VideoViewManager 中添加相关的接口，如常见的准备视频资源、视频播放状态、视频播放错误等。然后，通过 RCTEventEmitter 发送给 JS 端，JS 端封装组件实现对 Native 通知的监听，并提供给 JS 端调用。

8.3 小结

在原生移动应用开发中，系统为开发者提供了成百上千的 UI 组件。同样，React Native 也为开发者封装了大量最常用的组件，但是对于一些不常用的组件，系统并没有直接提供，这就需要开发者具备原生系统开发能力和组件封装能力，并将封装的组件提供给 React Native 使用。

学习本章，开发者需要具备一定的原生开发基础，通过本章，读者将更深层次地学习到原生开发的知识以及混合高级开发技能。学习本章，读者需要重点掌握 JS 调用原生组件的流程。

热更新与打包部署

由于 Apple 严格的审核标准和低效率，iOS 应用的发版速度极慢，这对于大多数团队来说是不能接受的，所以热更新对于 iOS 应用来说就显得尤其重要。而就在前不久，苹果出台了相关协议禁止 WaxPatch、JSPatch 等 JS 脚本注入方式的热修复框架，好在采用 React Native 方式的热更新似乎并没有受到多大影响。

对于 Android 应用开发来说，发布版本没有那么繁琐了，审核也没那么严格，上线就容易了很多。再加上最近两年陆续涌现了一大批热更新框架（如阿里的 Andfix、微信的 Tinker、QQ 空间的 Nuva 等），对于及时修复 Android 开发中出现的问题也是方便了不少。可以说，热更新是移动开发中必不可少的技能。

9.1 iOS应用打包

相对于 Android 的打包过程，iOS 打包显得有些繁琐，或许是 Apple 一贯追求极致的思想，其对于产品的质量要求更高。iOS 打包发布主要有以下几个步骤。

- ① 生成离线 Bundle 资源文件。
- ② 生成签名密钥。
- ③ 利用签名密钥生成 release 的 ipa 文件。
- ④ 发布到应用市场供用户下载安装。

9.1.1 iOS应用配置

添加应用配置

在正式打包以前，开发者需要对 iOS 项目的相关配置进行设置。在 Xcode 中打开项目工程，如图 9-1 所示。

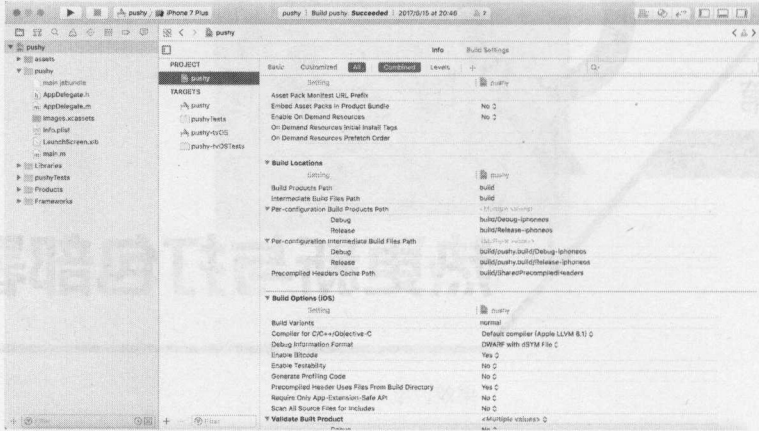


图9-1 Xcode应用配置

打开 Build Settings 面板，在本面板中主要有两个属性需要设置：Base SDK Version（基础 SDK 版本）和 iOS deployment target（iOS 部署版本）。其中，部署版本表示应用支持的最低运行版本，而基础 SDK 版本表示项目编译所依赖的 SDK 版本。

配置应用图标和启动图像

应用图标会展示在用户设备的主界面上，也会显示在 App Store 上。对于 iOS 开发者来说，只需要按照尺寸分类，将不同分辨率的尺寸添加到相应的分类中即可，应用在启动的时候会自动匹配对应尺寸的图片。配置应用图标如图 9-2 所示，配置应用启动图像如图 9-3 所示。

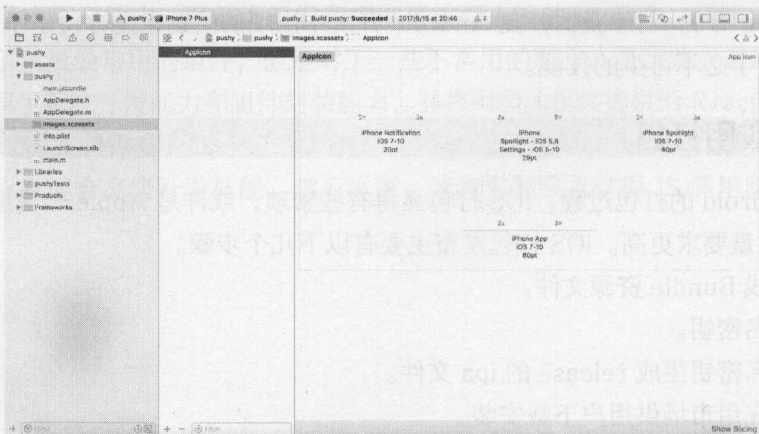


图9-2 配置应用图标

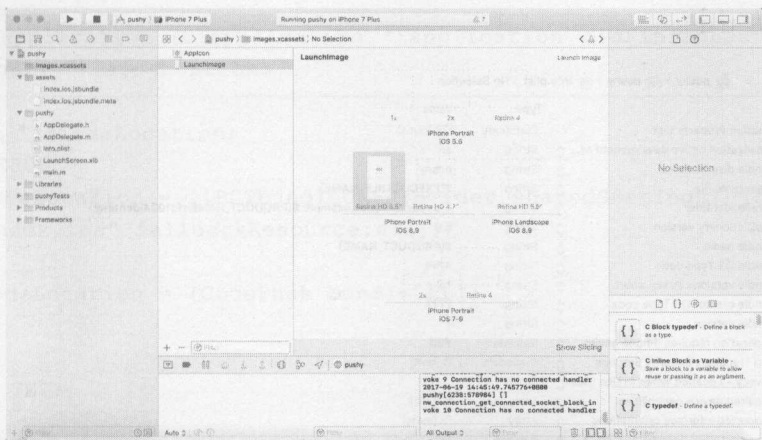


图9-3 配置应用启动图像

然后，删除之前安装的应用程序，重新安装就可以看到图标和启动图像了。图 9-4 所示是应用重新运行安装后的效果。



图9-4 应用启动图标

由于某些原因，应用图标或启动图像没有配置成功，这时请先检查【App Icon】和【Launch Images】菜单下的设置，或者切换到【General】菜单下确认是否可以找到相关图片。

配置网络白名单

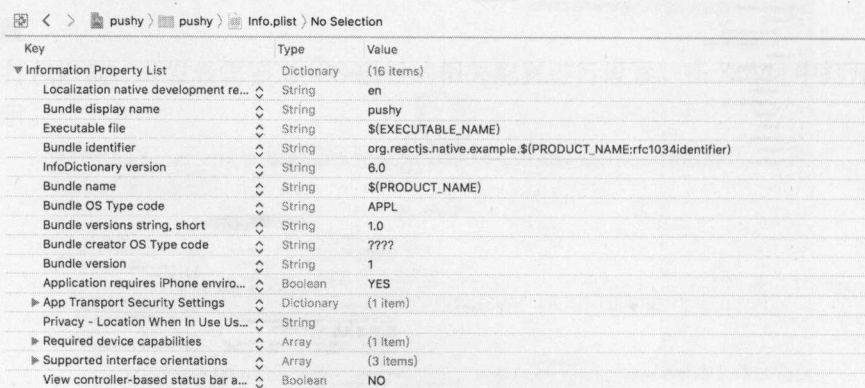
对于大多数应用 APP 而言，网络请求是必不可少的组成部分，如果你的应用还在使用 HTTP 方式请求数据，那么设置网络白名单是必须的，其配置界面如图 9-5 所示。但是，对于 iOS 9 及以上版本，非 HTTPS 的网络是被禁止的，在以前可以将 NSAllowsArbitraryLoads 设置为 YES 来禁用 ATS (Apple Terminal Service)，不过从 iOS 10 起，Apple 禁止通过这种方法跳过 ATS。但是，通过 NSExceptionDomains 来针对特定的域名开放 HTTP，可以很容易地通过审核。使用这种方式，每个域名都需要如下 3 个属性：

NSIncludesSubdomains：是否包含子域，设置为 true。

NSEnvironmentRequiresForwardSecrecy：指定域名是否支持 Forward Secrecy，设置为 false。

NSEnvironmentAllowsInsecureHTTPLoads：是否能使用 HTTP 协议，默认只能请求

HTTPS, 设置为 true。



The screenshot shows the 'Info.plist' file in Xcode. The 'Key' column lists various system properties, the 'Type' column shows their data types, and the 'Value' column shows the current values. The 'pushy' key is highlighted in the 'Value' column.

Key	Type	Value
Information Property List	Dictionary	(16 items)
Localization native development re...	String	en
Bundle display name	String	pushy
Executable file	String	\$(EXECUTABLE_NAME)
Bundle identifier	String	org.reactjs.native.example.\$(PRODUCT_NAME:rfc1034identifier)
InfoDictionary version	String	6.0
Bundle name	String	\$(PRODUCT_NAME)
Bundle OS Type code	String	APPL
Bundle versions string, short	String	1.0
Bundle creator OS Type code	String	????
Bundle version	String	1
Application requires iPhone enviro...	Boolean	YES
App Transport Security Settings	Dictionary	(1 item)
Privacy - Location When In Use Us...	String	
Required device capabilities	Array	(1 item)
Supported interface orientations	Array	(3 items)
View controller-based status bar a...	Boolean	NO

图9-5 配置网络白名单

9.1.2 打包离线Bundle

使用 Xcode 打开项目, 打开 AppDelegate.m 文件, 将下面这一行注释掉并新增一行代码:

```
// 注释掉这一行
/*jsCodeLocation = [[RCTBundleURLProvider sharedSettings] jsBundleURLForBundleRoot:@"index.ios" fallbackResource:nil];*/
// 新增一行
jsCodeLocation = [[NSBundle mainBundle] URLForResource:@"index.ios" withExtension:@"jsbundle"];
```

上述代码的作用是让 React native 直接使用新增加的 jsbundle 文件, 而不是从 index.ios 入口文件启动, 这样就摆脱了对本地 Node 环境的依赖。

将默认加载方式修改成 jsbundle 方式启动后, 需要对项目进行打包操作。具体的, 在 iOS 项目的根目录下新建一个 assets 文件, 然后使用如下的打包命令:

```
react-native bundle --entry-file index.ios.js --platform ios --dev false
--bundle-output ./ios/assets/index.ios.jsbundle --assets-dest ./ios/assets
```

离线 bundle 文件打包成功后, 将会在 iOS 项目文件下生成一个资源包文件, 该文件主要包含 index.ios.jsbundle 和 index.ios.jsbundle.meta 两部分, 如图 9-6 所示。

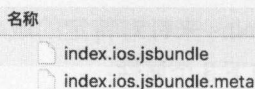


图9-6 iOS离线Bundle文件

如果在项目中使用了 CodePush 热更新框架, 也可以通过 CodePush 来读取本地的 jsbundle 文件。相关代码如下:

```

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
  (NSDictionary *)launchOptions
{
    NSURL *jsCodeLocation;
    #ifdef DEBUG
        jsCodeLocation = [[RCTBundleURLProvider sharedSettings] jsBundleURLForBundleRoot:@"index.ios" fallbackResource:nil];
    #else
        jsCodeLocation = [CodePush bundleURL];
    #endif
    ...
    return YES;
}

```

9.1.3 设置发布Scheme

正式打包前，我们需要将 Build Scheme 设置为 Release，而不是 Debug。依次选择【Product】→【Scheme】→【Edit Scheme】，打开【Scheme】菜单。修改【Build Configuration】为【Release】，并取消【Debug executable】的勾选，这意味着调试菜单会被关闭，如图 9-7 所示。



图9-7 修改Scheme信息

9.1.4 发布应用

iOS 应用开发完成之后，开发者需要将应用上传到 App Store 之后用户才能下载。但是在上传到 App Store 之前，你需要拥有一个开发者账号。以前申请一个开发者账号，开发者需要交纳 99 美金，现在如果是团队开发的项目，只需要该团队拥有一个开发者账号即可，团队成员可以共享这个开发者账号。如果你已经具备开发者的资格，那么直接创建建档后，将应用提交到 App Store 等待审核即可如图 9-8 所示。具体操作如下。

❶ 选择【Product】→【Archive】，如果【Archive】为灰色禁用状态，请选择真机作为构建目标。

❷ 如果成功，将会出现 Archive 界面，点击【Upload to App Store】按钮，应用就提交

到了 App Store。

③ 接下来，只需要耐心等待审核结果即可。

如果你还不是开发者，或者你想开发一个属于自己的应用，那么，打开 Apple 开发者中心页面 (<https://developer.apple.com/>)，按照相关的介绍申请一个属于自己的开发者账号吧。关于这方面的资料，请读者自行参考 Apple 提供给开发者的相关文档（官方地址：<https://developer.apple.com/develop/>）。

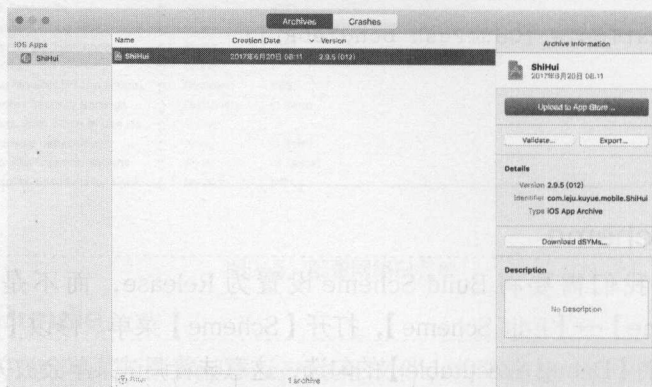


图9-8 创建Scheme，提交应用商店

9.2 Android应用打包

Android 要求所有应用都有一个数字签名才会被允许安装在用户手机上，所以在把应用发布到类似 Google Play store 这样的应用市场之前，你需要先生成一个签名的 APK 包。

对于 Android 来说打包主要有两种方式：一是借助于 Android Studio 的图形化界面进行打包；另一种是借助命令行进行打包操作。不过，需要注意的是，由于正式打包后，原生应用会加载离线 bundle 资源，所以在打平台包之前，需要先打离线 bundle 包。对于 Android 来说，打包发布主要有以下几个步骤。

- ① 生成离线 Bundle 资源文件。
- ② 生成签名密钥。
- ③ 利用签名密钥生成 release 的 APK 文件。
- ④ 发布到应用市场供用户下载安装。

9.2.1 打包离线Bundle

在 Android 项目的 src 的 main 目录下新建一个 assets 文件存放 bundle 资源；然后在工程根目录下执行打包命令。

Android 环境下打开 bundle 文件的命令如下：

```
react-native bundle --entry-file index.android.js --bundle-output ./android/app/src/main/assets/index.android.jsbundle --platform android --assets-dest ./android/app/src/main/res/ --dev false
```

参数说明如下。

- `entry-fil`: 指定入口文件。上面的命令指定React Native项目的`index.ios.js`作为入口。
- `bundle-output`: 指定输出的jsbundle文件路径和文件名。一定要先创建存放bundle的文件夹，不然会报文件夹找不到的错误。
- `platform`: 指定特定的平台类型。
- `assets-dest`: 指定资源文件夹路径。包含图片、node模块等资源文件。
- `dev`: 是否为开发模式。如果设置为`false`，bundle文件会被压缩。

关于其他的一些命令，如 `transformer`、`prepack`、`bundle-encoding` 等，读者可自行阅读官网的详细介绍。

离线 bundle 文件打包成功后，将会在 Android 的项目文件下生成 jsbundle 资源包文件，该文件主要包含 `index.ios.jsbundle` 和 `index.ios.jsbundle.meta` 两部分，如图 9-9 所示。

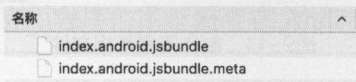


图9-9 Android离线Bundle文件

注意：在使用命令打包的过程中，如果提示文件路径错误，请确认项目目录下是否存在 `assets` 文件夹，文件的路径是否准确，其他的请根据错误提供操作。

9.2.2 生成签名密钥

Android 生成签名密钥有两种方式：命令方式和 Android Studio 界面方式。首先，介绍下使用命令方式制作签名密钥，命令如下：

```
keytool -genkey -v -keystore releasekey.keystore -alias key-alias -keyalg RSA -keysize 2048 -validity 10000
```

然后按照提示输入用户名和密码等信息即可。相比与命令方式，使用界面方式更加方便，使用 Android Studio 制作签名密钥主要有以下几个步骤。

❶ 打开 Android Studio，选择【Build】→【Generate Signed APK】，在打开的界面点击【Next】，如图 9-10 所示。



图9-10 制作签名密钥

- ② 打开选择【Create new...】，然后填写相关信息，最后点击创建按钮即可，如图9-11所示。

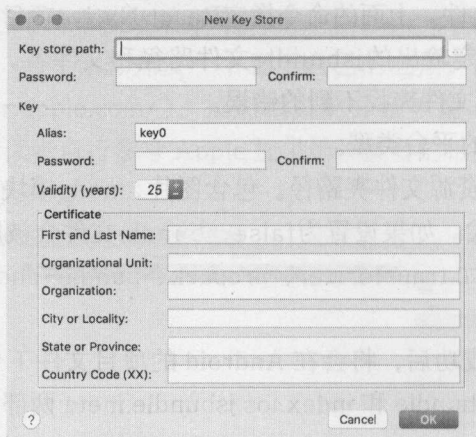


图9-11 制作签名密钥文件

9.2.3 生成签名APK

命令方式打包

在生成正式包之前，需要注意的是，如果你打算使用命令方式打包，那么需要对 gradle 文件进行一些设置，具体步骤如下。

- ① 将签名密钥文件 xx.keystore 放到项目的 android/app 文件夹下，新建或打开 .gradle/gradle.properties，添加如下代码：

```
MYAPP_RELEASE_STORE_FILE=my-release-key.keystore
MYAPP_RELEASE_KEY_ALIAS=my-key-alias
MYAPP_RELEASE_STORE_PASSWORD=*****
MYAPP_RELEASE_KEY_PASSWORD=*****
```

- ② 在项目的 app 目录下打开 gradle 配置文件，添加密钥相关配置：

```
android {
    defaultConfig { ... }
    signingConfigs {
        release {
            storeFile file(MYAPP_RELEASE_STORE_FILE)
            storePassword MYAPP_RELEASE_STORE_PASSWORD
            keyAlias MYAPP_RELEASE_KEY_ALIAS
            keyPassword MYAPP_RELEASE_KEY_PASSWORD
        }
    }
}
```

- ③ 进入 android 目录，使用如下命令生成签名 APK 文件：

```
./gradlew assembleRelease
```


说明：在 MacOS 和 Linux 系统中执行当前目录下的名为 gradlew 的脚本文件时，“./”是不可省略的，而在 windows 命令行下则需要去掉“./”。

Android Studio打包

另一种打包方式就是利用 Android Studio 进行打包，这种界面化的操作简单易懂，如图 9-12 所示。打开项目，依次选择【Build】→【Generate Signed APK】，输入相关信息，点击【Next】，配置正式 APK 输出路径（见图 9-13），然后等待系统生成正式的签名 APK 即可。APK 文件打包完成后，可以在 /android/app/build/outputs/apk 目录下找到生成的签名正式 APK 文件包。

在将应用发布到市场之前请先使用真机在本地模拟运行环境，查看是否可以正常运行。安装的时候，可以使用 gradlew installRelease 命令上传 APK 到物理设备上，也可以使用 USB 工具直接安装。测试无问题后将正式的 APK 发布到应用市场供用户下载安装即可。

配置应用图标和启动图像

对于 Android 开发者来说，应用程序的图标需要在 AndroidManifest.xml 添加相关配置。

```
<application
    android:name=".MainApplication"
    //...添加应用程序 icon
    android:icon="@mipmap/app_icon"
    //...
>
```

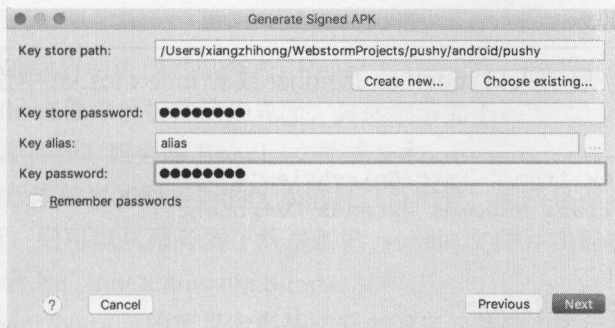


图9-12 使用Android Studio打包

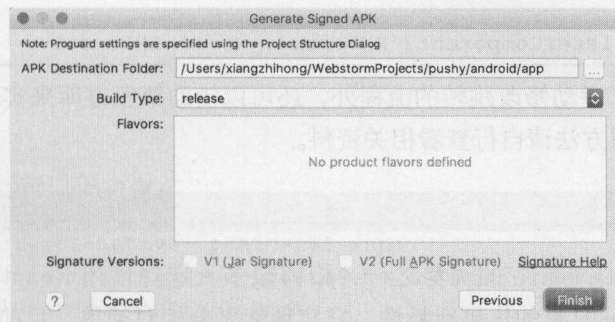


图9-13 配置APK输出位置

在配置闪屏页面方面，Android 没有默认的闪屏页面，如果需要配置启动页面，可以新建一个闪屏页面，并将闪屏页面作为默认启动页，将应用的默认启动页面修改为闪屏页。新增一个启动页面，在 Android 的入口组件（index.android.js）中将启动页面改为闪屏页面。例如，下面是闪屏页面代码：

```
export default class SplashView extends Component {
  // 倒计时 3 秒后进入首页
  componentDidMount() {
    setTimeout(() => {
      this.props.navigator.replace({
        // 跳转到主页面
        component: TabNavigatorView,
      });
    }, 3000);
  }
  render() {
    return (
      <View style={{ flex: 1 }}>
        <StatusBar hidden={true} />
        <Image style={styles.splash}
          source={require('../image/splash.png')} resizeMode={'cover'} />
      </View>
    )
  }
}
```

然后，在应用程序启动入口 index.android.js(或者 index.ios.js) 中设置下启动页面即可，相关代码如下：

```
import SplashView from '../src/SplashView';
export default class RNDemos extends Component {
  render() {
    return (
      <SplashView />
    );
  }
}
AppRegistry.registerComponent('RNDemos', () => RNDemos);
```

当然，开发者除了手动修改跳转的流程外，还可以借助第三方库来实现，例如 rn-splash-screen。读者可以使用方法请自行查看相关资料。

9.3 热更新

热更新作为 React Native 的优势之一，相信很多人选择使用 React Native 来开发应用，也是因为 React Native 具有的热更新特性，它实现起来会相对简单。目前，针对 React Native 提出的热更新方案中，比较成熟的是微软的 CodePush 和 React Native 的 pushy。由于 Apple

禁止了热更新行为，因此关于热更新的讲解主要以 Android 平台为主。

9.3.1 热更新原理

React Native 的热更新并不像原生应用更新那么复杂，因为 JavaScript 本身就是动态加载的，React Native 的热更新更像原生 APP 的版本更新。React Native 热更新流程如图 9-14 所示。

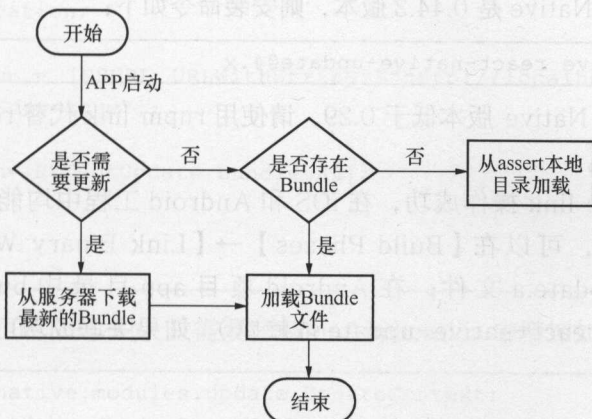


图9-14 热更新原理图

React Native 的加载启动流程可总结为：React Native 将一系列资源打包成 JS Bundle 文件，系统加载 js bundle 文件来解析并渲染界面，具体细节如下：

- ❶ 通过 Restful 服务获取服务端 Bundle 文件的版本。
- ❷ 将本地缓存的版本信息和服务器的版本信息进行比较，判断是否需要更新。
- ❸ 如果需要更新，则可以从服务器下载最新的 bundle 文件并实现更新即可。如果没有更新，则加载 assets 目录下的 index.android.bundle 文件。

当下选择使用 React Native 的项目大多是在原有项目的基础上进行接入的，所以，要在混合项目中使用热更新，JS 端还有很多工作需要做。

9.3.2 热更新配置

安装命令

在项目根目录下运行以下命令：

```

npm install -g react-native-update-cli rnpm
npm install --save react-native-update@ 具体版本请看下面的表格
react-native link react-native-update
  
```

React Native 对应的 react-native-update 版本信息如表 9-1 所示。

表 9-1

react native 版本	react-native-update 版本
0.26 以下	1.0.x
0.27 - 0.28	2.x
0.29 - 0.33	3.x
0.34 - 当前	4.x

当前笔者的 React Native 是 0.44.3 版本，则安装命令如下：

```
npm install --save react-native-update@4.x
```

注意：如果 React Native 版本低于 0.29，请使用 rnpm link 代替 react-native link 命令进行库文件的依赖关联。

如果 react-native link 操作成功，在 iOS 和 Android 工程中均能看到相应的依赖（具体表现为打开 Xcode，可以在【Build Phases】→【Link Binary With Libraries】依赖库看到 libRCTHotUpdate.a 文件；在 Android 项目 app 目录中 build.gradle 文件下的 dependencies 中看到 react-native-update 依赖库）。如果关联成功可以跳过下一步，否则需要手动关联。

手动关联

如果使用命令方式关联依赖库失败，可以使用下面的方式手动关联。

iOS：iOS 添加依赖库主要有以下几个步骤。

- ❶ 打开 Xcode，项目上右键点击【Libraries】→【Add Files to 工程名】。
- ❷ 进入【node_modules】→【react-native-update】→【ios】，选中 RCTHotUpdate。
- ❸ 选中项目，依次选择【Build Phases】→【Link Binary With Libraries】，然后添加 libRCTHotUpdate.a 依赖库。
- ❹ 在 Build Settings 里搜索 Header Search Path，然后重新编译一下项目即可。

Android：Android 关联依赖库主要有以下几个步骤。

- ❶ 在 android/settings.gradle 中添加如下代码：

```
include ':react-native-update'
project(':react-native-update').projectDir = new File(rootProject.projectDir,
'../node_modules/react-native-update/android')
```

- ❷ 在 android/app/build.gradle 的 dependencies 部分增加如下代码：

```
compile project(':react-native-update')
```

- ❸ 打开 MainActivity 文件，在 getPackages() 增加一行代码：

```
new UpdatePackage()
```

配置Bundle URL

配置 iOS 的 Bundle URL 时, 需要在工程目录下依次选择【Build Phases】→【Link Binary with Libraries】, 加入 libz.tbd 和 libbz2.1.0.tbd 等库。然后在 AppDelegate.m 文件中增加如下代码:

```
#import "RCTHotUpdate.h"

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSURL *jsCodeLocation;
    #if DEBUG
        jsCodeLocation = [NSURL URLWithString:@"http://localhost:8081/index.ios.bundle?platform=ios&dev=true"];
    #else
        jsCodeLocation=[RCTHotUpdate bundleURL];
    #endif
    // ... 其他代码
}
```

配置 Android Bundle URL 时, 只需要在 MainApplication 中增加如下代码:

```
import cn.reactnative.modules.update.UpdateContext;
private final ReactNativeHost mReactNativeHost = new ReactNativeHost(this) {
    @Override
    protected String getJSBundleFile() {
        return UpdateContext.getBundleUrl(MainApplication.this);
    }
    // ... 其他代码
}
```

iOS的ATS例外配置

从 iOS9 版本开始, Apple 要求开发者以使用 HTTPS 的方式进行数据请求, 以加快开发者部署 HTTPS 安全协议, 但是, 在过渡期使用 HTTP 方式的数据请求也不是完全禁止的。如果服务器使用的是 HTTP 方式, 需要在 info.plist 配置中添加如下代码:

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSExceptionDomains</key>
    <dict>
        <key>reactnative.cn</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <true/>
        </dict>
    </dict>
```

```
</dict>
</dict>
```

9.3.3 登录与创建应用

配置完基本信息，接下来，需要登录 Pushy 官网注册应用（官网地址：<http://update.reactnative.cn>）。然后在项目根目录下运行以下命令：

```
pushy login
email: <输入你的注册邮箱>
password: <输入你的密码>
```

登录成功后，项目文件夹下会多一个 .update 文件。如果项目涉及多人开发，为了安全，请不要把敏感配置文件上传到 Git 仓库中，可以在 .gitignore 文件的末尾增加一行 .update 来忽略这个文件。接下来，使用命令分别针对 Android 和 iOS 平台创建应用。

```
// 创建 iOS 应用
pushy createApp --platform ios
App Name: <输入应用名字>
// 创建 Android 应用
pushy createApp --platform android
App Name: <输入应用名字>
```

如果你已经通过浏览器的方式创建了应用，可以使用命令直接选择应用。命令如下：

```
pushy selectApp --platform ios
// 应用信息
Total 1 android apps
Enter appId:<输入应用编号>
```

使用上面的命令后，系统会在服务器端列出应用的相关信息。输入应用编号后，读者可以在文件夹下看到 update.json 文件，内容如下：

```
{
  "android": {
    "appId": 8482,
    "appKey": "3w1x_wfSMXNUdtMVYSXV3j9XhPBhArt6"
  },
  "ios": {
    "appId": 8483,
    "appKey": "Ri6eNMS3e5S6k1L903rOv8vq0qVPBSH_"
  }
}
```

注意：如果是多人协作开发，可以将 update.json 文件上传到 Git 等 CVS 系统上，不过对于账号等敏感信息请注意相关的信息安全。

9.3.4 添加热更新功能

为了在客户端中使用热更新功能，还需要在 React Native 端编写相关代码。实现热更新功能主要有以下步骤。

获取appKey

在检查更新时必须提供 appKey 来对比版本差异，这些信息被保存在 update.json 文件中。可以使用如下代码进行获取：

```
import {
  Platform,
} from 'react-native';
import _updateConfig from './update.json';
const {appKey} = _updateConfig[Platform.OS];
```

如果使用 pushy 命令，也可以在网页端直接查看到两个应用 appKey，并根据不同平台来进行选择。

检查与下载更新

定义异步函数 checkUpdate 检查当前版本是否需要更新。

```
checkUpdate(appKey)
  .then(info => {
  })
```

info 返回值主要有 3 种情况。

- {expired: true}: 应用包已过期，需要前往应用市场下载新的版本。
- {upToDate: true}: 当前已是最新，无需进行更新。
- {update: true}: 有新版本可以更新。info 内容主要包含有 name、description 和 metaInfo 等字段。下载的时候将 info 对象传递给 downloadUpdate 作为参数即可。

切换版本

downloadUpdate 的返回值是一个 hash 字符串，它是当前版本的唯一标识。可以使用 pushy 提供的 switchVersion 方法来立即切换版本（此时应用会立即重新加载），或者调用 switchVersionLater 方法让应用在下次启动的时候加载新版本。

首次启动、回滚

系统定义了一个 isFirstTime 常量用来记录是否进行了热更新操作，在应用每次更新完毕后的首次启动时，isFirstTime 常量会为 true。开发者可以在应用退出前合适的任何时机，调用 markSuccess 函数来改变更新的状态，否则应用下次启动的时候将会进行回滚操作。这一机制被称为“反触发”，当应用启动初期遭遇问题的时候，可以在下次启动时恢复运作。回滚操作的完整代码如下：

```
import React, {Component} from 'react';
import {
```

```

AppRegistry,
StyleSheet,
Platform,
Text,
View,
Alert,
TouchableOpacity,
Linking,
} from 'react-native';
import {
  isFirstTime,
  isRolledBack,
  packageVersion,
  currentVersion,
  checkUpdate,
  downloadUpdate,
  switchVersion,
  switchVersionLater,
  markSuccess,
} from 'react-native-update';

class HotFix extends Component {

  componentWillMount() {
    if (isFirstTime) {
      Alert.alert('提示', '这是当前版本第一次启动, 是否要模拟启动失败? 失败将回滚到
上一版本', [
        {text: '是', onPress: ()=>{throw new Error('模拟启动失败, 请重启
应用')}}),
        {text: '否', onPress: ()=>{markSuccess()}}],
      );
    } else if (isRolledBack) {
      Alert.alert('提示', '刚刚更新失败了, 版本被回滚. ');
    }
  }

  doUpdate = info => {
    downloadUpdate(info).then(hash => {
      Alert.alert('提示', '下载完毕, 是否重启应用?', [
        {text: '是', onPress: ()=>{switchVersion(hash);}},
        {text: '否', },
        {text: '下次启动时', onPress: ()=>{switchVersionLater(hash);}},
      ]);
    }).catch(err => {
      Alert.alert('提示', '更新失败. ');
    });
  };

  checkUpdate = () => {

```

```

    checkUpdate(appKey).then(info => {
      if (info.expired) {
        Alert.alert('提示', '您的应用版本已更新, 请前往应用商店下载新的版本', [
          {text: '确定', onPress: ()=>{info.downloadUrl && Linking.
openURL(info.downloadUrl)}}],
          []);
      } else if (info.upToDate) {
        Alert.alert('提示', '您的应用版本已是最新. ');
      } else {
        Alert.alert('提示', '检查到新的版本 '+info.name+', 是否下载? \n'+
info.description, [
          {text: '是', onPress: ()=>{this.doUpdate(info)}},
          {text: '否', },
        ]);
      }
    }).catch(err => {
      Alert.alert('提示', '更新失败. ');
    });
  };

  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          欢迎使用热更新服务
        </Text>
        <Text style={styles.instructions}>
          这是版本一 {'\n'}
          当前包版本号: {packageVersion}{'\n'}
          当前版本Hash: {currentVersion||' '(空)}{'\n'}
        </Text>
        <TouchableOpacity onPress={this.checkUpdate}>
          <Text style={styles.instructions}>
            点击这里检查更新
          </Text>
        </TouchableOpacity>
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
});

```



```

welcome: {
  fontSize: 20,
  textAlign: 'center',
  margin: 10,
},
instructions: {
  textAlign: 'center',
  color: '#333333',
  marginBottom: 5,
},
});

export default HotFix;

```

经过上面的操作，应用已经具备了热更新的能力。接下来，需要讨论一下热更新在正式环境的布置问题。

9.3.5 发布热更新版本

从上传发布版本到发布版本正式上线期间，不要修改任何脚本和资源，否则会影响更新时本地代码的获取，从而导致热更新失败。需要注意的是，在打包之前，首先需要使用命令生成 jsbundle 文件，成功后会在 Android 的 assets 和 iOS 的资源目录下生成 jsbundle 文件。命令如下：

```

react-native bundle --entry-file index.android.js --bundle-output ./android/
app/src/main/assets/index.android.jsbundle --platform android --assets-dest ./
android/app/src/main/res/ --dev false

```

发布iOS应用

发布 iOS 应用之前，按照正常的打包流程打包 .ipa 文件，在 Xcode 中选择运行设备为真机或 Generic iOS Device，然后在菜单中依次选择【Product】→【Archive】，运行如下命令：

```
pushy uploadIpa <your-package.ipa>
```

接下来，直接在 Xcode 中使用上传按钮将应用上传到 AppStore 中，当应用重新启动之后，就会调用代码执行更新操作。

发布Android应用

使用 AndroidStudio 按照正常的打包流程生成 apk 文件，或者直接使用打包命令“gradlew assembleRelease”生成 apk 文件。然后就可以在 android/app/build/outputs/apk/app-release.apk 中找到对应的签名包。

```
D:\react\workspace\pushy\android>gradlew assembleReleas
```

使用如下命令将 APK 上传到 pushy 服务器上以供版本更新对比使用。

```
pushy uploadApk android/app/build/outputs/apk/app-release.apk
```

上传成功后，打开 pushy 会看到上传的结果，如图 9-15 所示。当然，还可以将 APK 包发布到各大应用市场以供大家下载。需要注意的是，在实际操作中可能会遇到很多问题，读者请根据实际情况解决。

发布新的热更新版本

为了验证热更新的效果，笔者将源代码做了简单的修改（如打印一段日志），然后生成一个新的热更新版本。然后，使用命令发布新的版本，具体如下：

```
pushy bundle --platform <ios|android>
Bundling with React Native version: 0.45.1
<各种进度输出>
Bundled saved to: build/output/android.1459850548545.ppk
Would you like to publish it?(Y/N)
  Uploading [=====] 100%
0.0s
Enter version name: <输入版本名字，如 1.0.0-rc>
Enter description: <输入版本描述>
Enter meta info: {"ok":1}
Ok.
Would you like to bind packages to this version?(Y/N)
```

如果想要立即发布，此时输入 Y。当然，也可以使用命令“pushy publish --platform <ios|android> <ppkFile>”来发布版本。如果想要给指定的平台下发热更新功能，提醒对应平台的用户进行更新。可以使用如下命令：

```
pushy update --platform <ios|android>
```

除了使用命令的方式外，系统还允许开发者使用浏览器方式直接上传热更新包，当客户端应用程序再次启动的时候，客户端就可以进行检查更新操作了。

9.4 小结

本章主要介绍了 Android 平台、iOS 平台打包上线流程。Android 审核比较宽松，但是平台较多，可能需要针对不同的平台做一些适配优化。而 iOS 上线相对比较繁琐，审核也更加严格。热更新技术是现在移动应用开发中比较热门的技术，也是每个开发者必须掌握的核心技术。本章主要从热更新的原理出发，介绍了热更新实例，为及时修复线上问题提供了技术支持。

基于LBS的天气预报应用开发

Windows 系统的诞生成就了微软帝国，同时也造就了 PC 时代的繁荣。如今，以 Android 和 iPhone 手机为代表的智能移动设备的出现，为移动互联网技术的发展带来了无限的动力。

科技的进步会给人们的生活带来方便和快捷，随着科技的发展，移动智能终端逐渐走进人们的视线，相关应用越来越广泛，并在人们的日常生活中扮演着越来越重要的角色。因此，关键应用程序的开发成为影响移动智能终端普及的重要因素，设计并开发实用、方便的应用程序具有重要的意义和良好的市场前景。

近几年来，随着移动技术的成熟和智能手机的普及，移动应用的需求与日俱增，移动应用开发成为当下最热门的技术之一。

10.1 需求分析与确定

软件开发是根据用户需求，建造软件系统或者部分系统的过程，涉及了多学科、多领域。软件开发通常包含：需求分析、功能和实现的算法分析、总体结构设计和模块设计、编程和调试、程序联调、测试和提交程序等多个步骤。

10.1.1 需求分析

随着气象现代化工作的推进，气象灾害监测预警能力的提升，基于移动互联网平台的天气预报应用是气象突发事件信息快捷高效发布的重要途径之一，也是报纸、电台、电视、声讯、

互联网等各类传统媒体发布渠道的补充。

天气预报系统作为一个基于位置服务的综合应用系统,涉及多个学科和知识,其架构如图 10-1 所示。总体上看,基于 LBS 的系统可以看成是由移动通信网络和计算机网络结合而成的。移动终端通过移动通信网络发出请求,经过网关传递给 LBS 服务平台;服务平台根据用户请求和用户当前位置进行处理,并将处理的结果通过网络返回给移动终端。

基于移动互联网的天气预报 APP 系统是一种分布式系统,采用客户端/服务器体系。该体系结构主要由表示层、逻辑层和数据层三部分组成。

其中,表示层是基于 WAP、Web、J2ME 等技术开发的客户端 APP 应用。逻辑层包括 Web 服务器、GIS 应用服务器、无线网关、移动定位网关等系统。数据层主要是空间数据和属性数据的数据库管理系统。其系统逻辑结构如图 10-2 所示。

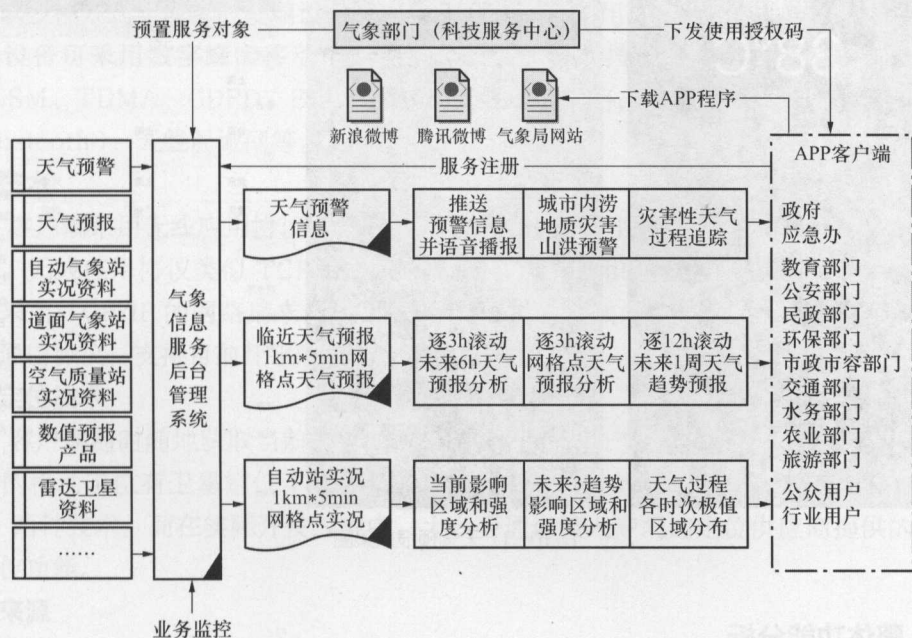


图 10-1 天气预报架构图

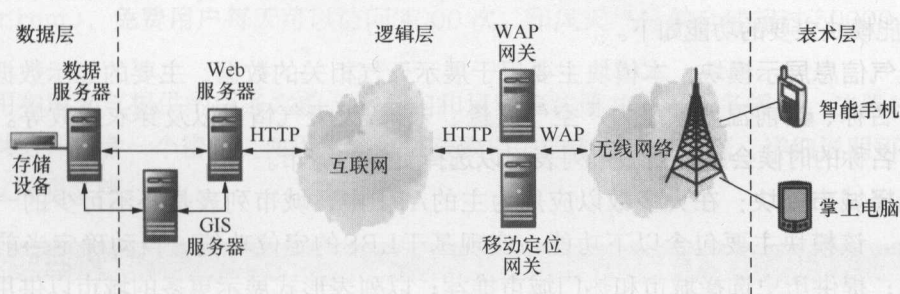


图 10-2 天气预报系统逻辑示意图

10.1.2 需求确定

商业天气预报系统，涉及的学科和知识很多，是一个复杂的大型系统。对于移动应用开发工程师来说，他们并不需要知道整个系统的架构，而是只需要向服务器发送请求，然后绘制客户端的页面即可。基于这种情况，客户端的主要提供以下功能：

- 根据系统定位，获取当前城市未来几天内的实时天气数据并填充页面。
- 提供城市选择功能，获取选择城市的天气数据。
- 系统要具备稳定性和实时性，保证获取的数据都是最新的。

为了方便需求的理解，首先看一下项目的最终运行效果图如图 10-3 所示。

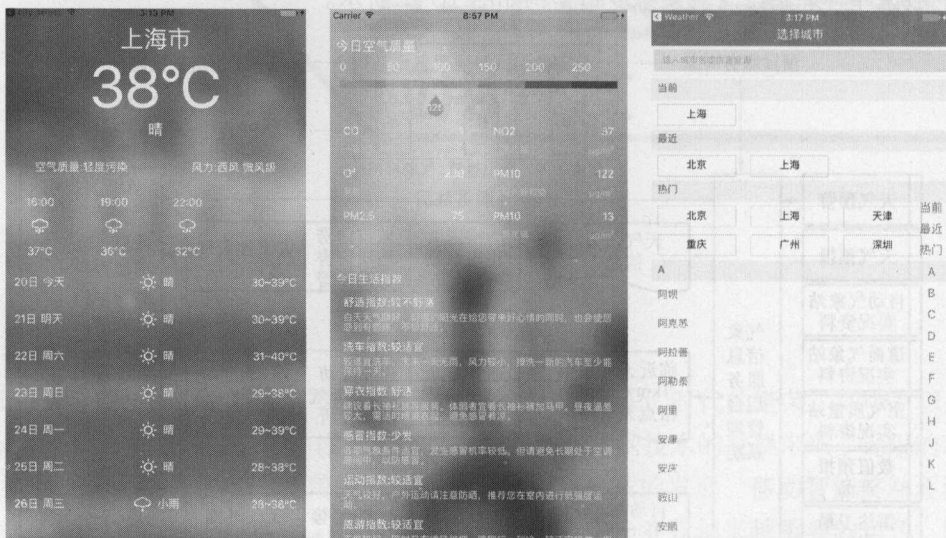


图10-3 天气预报效果图

10.1.3 整体功能分析

根据需求分析可知，本天气预报应用主要包含两个模块：天气信息展示，以及城市列表选择。各功能模块主要的功能如下。

- **天气信息展示模块：**本模块主要用于展示天气相关的数据，主要的显示数据包括：城市名称、当前温度、风力、空气质量、未来7天天气情况以及穿衣指数等。当点击城市名称的时候会跳转到城市列表，以选择更多的城市。
- **选择城市模块：**在大多数以应用为主的APP中，城市列表是必不可少的一个功能模块。该模块主要包含以下功能：实现基于LBS的定位功能，自动确定当前用户所在地；提供历史选择城市和热门城市推荐；以列表形式展示更多的城市以供用户选择；同时为了方便用户查找，支持以输入框的形式来过滤查询预报城市。

- 桌面小部件：为了方便用户实时了解天气状况，Android系统桌面上会生成一个快捷的小部件，点击小部件就会唤醒应用程序。

10.1.4 技术与架构分析

一个完整的天气预报系统会涉及很多技术。当然，对于移动客户端开发来说，我们并不需要了解服务器架构，移动开发者只需要向服务器发送请求，然后根据返回的数据绘制本地界面即可。

为了方便讲解，下文罗列了一些与移动技术相关的一些技术要点以供参考。

移动接入技术

移动互联网是 APP 客户端表述层与服务器端进行通信和数据交互的网络运行平台。移动接入技术能以多种方式实现智能终端设备接入移动互联网，使用户摆脱线缆和位置的束缚。目前，智能终端设备可采用数字蜂窝移动电话网络（CPN）接入技术接入移动互联网，如 CDMA、GPRS、GSM、TDMA、CDPD、EPGE 等多种无线承载网络；或是采用局域网的接入技术，如蓝牙（Bluetooth）、无线局域网等。

移动访问技术

APP 客户端采用无线应用协议（WAP）或采用 TCP/IP 协议从移动互联网获取所需的气象信息服务。无线应用协议类似 TCP/IP，以标记语言 WML 和脚本语言 WMLScript 处理 WAP 网页。而采用 TCP/IP 的网络请求通过诸如 json 格式进行数据的交互。WAP 作为一种全球开放的无线通信协议，支持目前几乎所有的移动设备。

移动终端定位技术

APP 客户端随时随地获取当前位置的天气信息服务依赖于终端设备的移动定位技术。智能终端设备的移动定位有卫星定位（GPS、A-GPS）、移动通信网络定位（COO、TOA、AOA、E-OTD）两种技术。而在实际开发过程中，大多会通过接入专门的定位供应商提供的相关技术来实现定位功能。

天气数据来源

对于天气数据的来源，我们采用网络上开放的免费 API。中国天气网、新浪网等，都提供免费的 API 服务。本例中，我们采用和风天气提供的免费 API（官网地址：<https://www.heweather.com>），免费用户每天可以访问 4000 次。和风天气提供全球超过 50000 个城市的天气数据。

在使用和风天气提供的服务之前，需要向和风天气注册并获得服务授权，注册成功之后，系统会提供给开发者一个密钥，使用它就可以获取相关的天气接口数据。详细说明可以参考和风天气开发说明文档。

10.2 项目设计

通过前面的需求分析和技术分析发现，本示例项目主要由两个页面组成：天气展示页面和

城市列表页面，其系统架构模型大体如图 10-4 所示。

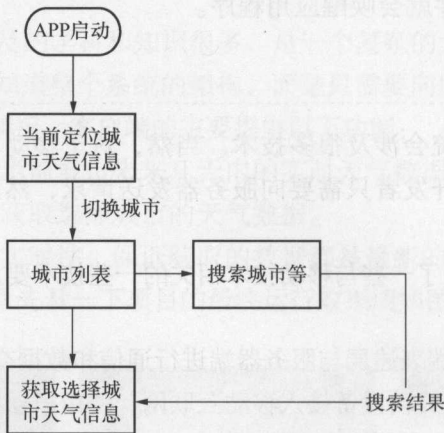


图10-4 天气预报项目架构示意图

项目代码完整结构如图 10-5 所示。



图10-5 项目代码结构图

其中，index.ios.js 是 iOS 程序的入口，index.android.js 是 Android 程序的入口，src 为程序源码的存放位置，package.json 是程序依赖的一些第三方库和配置信息。package.json 依

依赖的第三方库如下：

```

"dependencies": {
  "mobx": "^3.2.1",
  "mobx-react": "^4.2.2",
  "react-native": "0.44.3",
  "react-native-city-select": "^0.1.0",
  "react-native-deprecated-custom-components": "^0.1.1",
  "react-native-drawer-layout": "^1.3.2",
  "react-native-storage": "^0.2.2",
  "react-native-swipeout": "^2.1.1",
  "react-native-vector-icons": "^4.0.0",
  "react-navigation": "^1.0.0-beta.11"
}
//...其他系统库

```

在本项目中，使用 Mobx 作为状态管理工具类，在使用前，需要对 Mobx 环境进行简单的配置。在项目目录下创建一个 .babelrc 文件，或者使用如下命令创建：

```

npm i babel-plugin-transform-decorators-legacy babel-preset-react-native-stage-0 --save-dev

```

在项目目录下找到 .babelrc 文件，打开后添加如下内容：

```

{
  "presets": ["react-native"],
  "plugins": ["transform-decorators-legacy"]
}

```

10.3 程序入口与工具模块

作为一个全新的项目，在需求分析和项目评审做完之后，就到了编码阶段。

10.3.1 程序入口

当项目创建后，就可以编写入口文件了。对于 Android 来说入口文件就是 index.android.js，对于 iOS 来说是 index.ios.js。

从整体软件架构层次出发，为了方便对程序入口的管理和维护不建议直接使用默认的 AppRegistry.registerComponent 注册某个入口组件来启动应用。由于本项目使用 react-navigation 来作为路由跳转的方案，为了统一管理新建一个 Navigation.js 文件来维护页面堆栈。具体代码如下：

```

Navigation.js
import {StackNavigator} from 'react-navigation'

import WeatherScreen from '../containers/WeatherScreen'

```



```
import CityScreen from '../containers/SelectCityScreen';
```

```
const Navigation = StackNavigator({
  WeatherScreen: {screen: WeatherScreen},
  CityScreen: {screen: CityScreen},
```

```
});
```

```
export default Navigation
```

```
WeatherScreen.js
```

```
/**
```

```
 * 天气预报主页面
```

```
 */
```

```
'use strict';
```

```
import React, {Component} from 'react'
```

```
import WeatherIndex from '../ui/WeatherIndex'
```

```
export default class WeatherScreen extends Component {
```

```
  static navigationOptions = {
```

```
    title: null,
```

```
    header: null,
```

```
  }
```

```
  constructor(props) {
```

```
    super(props);
```

```
    this.state = {};
```

```
  }
```

```
  render() {
```

```
    return (
```

```
      <WeatherIndex navigation={this.props.navigation}/>
```

```
    )
```

```
  }
```

```
}
```

```
CityScreen.js
```

```
/**
```

```
 ** 城市选择主页面
```

```
 */
```

```
'use strict';
```

```
import React, {Component, PropTypes} from 'react'
```

```
import SelectCity from '../ui/SelectCity'
```

```
export default class SelectCityScreen extends Component {
```

```

static navigationOptions = {
  title: '选择城市',
  headerStyle: {
    backgroundColor: '#e75404'
  },
  headerTintColor: 'white'
}

constructor(props) {
  super(props);
  this.state = {};
}

render() {
  return (
    <SelectCity/>
  )
}
}

```

react-navigation 库提供 3 种导航样式，分别为 StackNavigator、TabNavigator 和 DrawerNavigator。其中 StackNavigator 的作用类似于屏幕上方导航栏，使用的时候只需要将页面放到 StackNavigator 内即可。由于 StackNavigator 维护的是类似于栈的数据结构，所以 StackNavigator 里面第一个页面在栈的顶端。

然后我们定义一个中间配置文件 Root.js。在这里进行一些变量的设置，从而方便我们在不同环境下对日志、错误信息的读取。Root.js 相关代码如下：

```

import Navigation from './navigation/Navigation'

if (!__DEV__) {
  global.console = {
    info: () => {
    },
    log: () => {
    },
    warn: () => {
    },
    error: () => {
    }
  };
}

export default Navigation;

```

最后分别在 index.android.js 和 index.ios.js 文件注册这个文件即可。相关代码如下：

```
import { AppRegistry } from 'react-native';
import Root from './src/Root'

global.__APP__=true;
global.__ANDORID__=true;//Android 环境为 true
global.__IOS__=false;// Android 环境为 false

AppRegistry.registerComponent('Weather', () => Root);
index.ios.js
import { AppRegistry } from 'react-native';
import Root from './src/Root'

global.__APP__=true;
global.__ANDORID__=false; //ios 环境为 false
global.__IOS__=true; //ios 环境为 true

AppRegistry.registerComponent('Weather', () => Root);
```

在很多大型项目中，Android 平台和 iOS 平台会有很多差异，对于只在某个平台上才能运行的代码，一般通过变量来切换不同的代码环境。

10.3.2 数据模型定义与数据解析

在软件设计典范中，有一种独有的代码组织方式：MVC 框架。MVC 框架将业务逻辑、数据、界面显示分离，使代码层次更加清晰。而 M 通常表示为业务模型层，也就是定义的数据模型，在面向对象语言中，大多操作的也就是这些数据实体对象。

在本项目中，当我们向和风天气发出天气数据请求后（请求地址：<https://free-api.heweather.com/v5/weather?key=19713447578c4afe8c12a351d46ea922&city=上海>），得到如下的 json 返回数据。

```
{ HeWeather5: [{
  aqi: {},
  basic: {},
  daily_forecast: [],
  hourly_forecast: [],
  now: {},
  status: "ok",
  suggestion: {}
}]}
```

同时，为了方便城市切换数据的请求，需要将天气数据的请求封装成一个可以调用的方法。相关代码如下：

```

/**
 * 根据城市名获取天气
 * @param name
 */
requestWeatherByName(name) {
  this.loading = true;
  return fetch("https://free-api.heweather.com/v5/weather?key=19713447578c4afe8c12a351d46ea922&city=" + name)
    .then((response) => {
      if (response.ok) {
        return response.json();
      }
    })
    .then((jsonData) => {
      this.saveWeatherData(jsonData);
      this.loading = false;
    }).done();
}

```

当数据返回之后，我们需要将结果映射到定义的字段上，综合页面的展示和接口数据返回的情况，先定义 4 个实体对象文件：天气数据（WeatherInfo.js）、未来一周天气情况预测列表（WeatherItemInfo.js）、空气质量（AirItemInfo.js）和生活指数（SuggestionItemInfo.js）。具体代码如下：

```

/**
 * 天气情况
 */
'use strict';
import {observable, computed} from 'mobx';

export default class Weather {
  @observable aqi;
  @observable basic;
  @observable daily;
  @observable hourly;
  @observable now;
  @observable suggestion;

  constructor(jsonData) {
    this.aqi = jsonData.aqi;
    this.basic = jsonData.basic;
    this.daily = jsonData.daily_forecast;
    this.hourly = jsonData.hourly_forecast;

```

```

        this.now = jsonData.now;
        this.suggestion = jsonData.suggestion;
    }
}

```

WeatherItemInfo.js

```

/**
 * 一周天气情况预测 Item
 */
'use strict';
import {observable, computed} from 'mobx';

export default class WeatherItemItem {
    @observable cityName; // 城市名称
    @observable tmp; // 温度区间, 如 30 ~ 40℃
    @observable iconUrl; // 天气图标

    constructor(cityName, tmp, iconUrl) {
        this.cityName = cityName;
        this.tmp = tmp;
        this.iconUrl = iconUrl;
    }
}

```

AirItemInfo.js

```

/**
 * 空气质量指数
 */
'use strict';
import {observable, computed} from 'mobx';

export default class AirItemInfo {
    @observable eng_name; // 天气质量参数名称, 如 CO、PM2.5
    @observable value; // 天气质量对应的值
    @observable chn_name; // 中文描述
    @observable unit; // 归类

    constructor(eng_name, value, chn_name, unit) {
        this.eng_name = eng_name;
        this.value = value;
        this.chn_name = chn_name;
        this.unit = unit;
    }
}

```

SuggestionItemInfo.js 是出行指数的实体类, 相关代码如下:

```

/**
 * 建议出行指数
 */
'use strict';
import {observable, computed} from 'mobx';

export default class SuggestionItemInfo{
  @observable type; // 建议类型
  @observable brf; // 建议指数标题
  @observable txt; // 建议指数描述文本

  constructor(type, brf, txt) {
    this.type = type;
    this.brf = brf;
    this.txt = txt;
  }
}

```

在原生 Android、iOS 开发中，当数据模型定义好后，我们可以直接使用 json 工具将字段的值直接映射到定义的数据模型上（Android 的 Gson，iOS 的 JSONKit）。在本示例中，我们并没有采用这种粗放的方式，而是根据实际情况，解析有用的字段到相应字段上。手动解析涉及的相关代码如下：

```

/**
 * 持久化天气数据并解析到数据模型
 */
saveWeatherData(jsonData) {
  let weatherData = jsonData.HeWeather5[0];
  this.weatherMap.set(weatherData.basic.city, new Weather(weatherData));
  this.convertAqiToList(weatherData);
  this.convertSuggestionList(weatherData);
  this.saveCityWeatherItem(weatherData);
}

// 未来一周天气情况
saveCityWeatherItem(weatherData) {
  let flag = -1;
  for (let i = 0; i < stateStore.cityList.length; i++) {
    if (stateStore.cityList[i].cityName == weatherData.basic.city) {
      flag = i;
      break;
    }
  }
  let weatherItem = new CityItemInfo(weatherData.basic.city,
    weatherData.daily_forecast[0].tmp.min + '~' + weatherData.daily_

```

```

forecast[0].tmp.max + '℃ ',
    ApiConfig.iconApi + weatherData.daily_forecast[0].cond.code_d +
    '.png');

    if (flag != -1) {
        stateStore.cityList[flag] = weatherItem;
    } else {
        stateStore.cityList.push(weatherItem);
    }
    stateStore.saveLocalCityData();
}

// 空气质量指数
convertAqiToList(weatherData) {
    this.aqiList = [];
    let aqi = weatherData.aqi.city;
    this.aqiList.push(new AirItemInfo('CO', aqi.co, '一氧化碳', 'mg/m³'));
    this.aqiList.push(new AirItemInfo('NO2', aqi.no2, '二氧化氮', 'μg/m³'));
    this.aqiList.push(new AirItemInfo('O3', aqi.o3, '臭氧', 'μg/m³'));
    this.aqiList.push(new AirItemInfo('PM10', aqi.pm10, '可吸入颗粒物', 'μg/
m²'));
    this.aqiList.push(new AirItemInfo('PM2.5', aqi.pm25, '可入肺颗粒', 'μg/
m³'));
    this.aqiList.push(new AirItemInfo('PM10', aqi.so2, '二氧化硫', 'μg/m³'));
}

// 生活指数
convertSuggestionList(weatherData) {
    this.lifeList = [];
    let suggestion = weatherData.suggestion;
    this.lifeList.push(new SuggestionItemInfo('舒适指数', suggestion.
comf.brf, suggestion.comf.txt));
    this.lifeList.push(new SuggestionItemInfo('洗车指数',
suggestion.cw.brf, suggestion.cw.txt));
    this.lifeList.push(new SuggestionItemInfo('穿衣指数', suggestion.
drsg.brf, suggestion.drsg.txt));
    this.lifeList.push(new SuggestionItemInfo('感冒指数',
suggestion.flu.brf, suggestion.flu.txt));
    this.lifeList.push(new SuggestionItemInfo('运动指数', suggestion.
sport.brf, suggestion.sport.txt));
    this.lifeList.push(new SuggestionItemInfo('旅游指数', suggestion.
trav.brf, suggestion.trav.txt));
    this.lifeList.push(new SuggestionItemInfo('紫外线指数', suggestion.
uv.brf, suggestion.uv.txt));
}

```

10.3.3 数据存储

在 React Native 开发中，系统为开发者提供了一个轻量级的本地数据存储方案：AsyncStorage。而在实际项目中，大多使用一个第三方库：react-native-storage，这是一个对本地持久存储方案的封装，它同时支持 React Native（AsyncStorage）和浏览器（localStorage）的数据存储。

使用 react-native-storage 时，应注意以下几点。

初始化

在使用 react-native-storage 时需要对它进行初始化，然后才能使用。初始化里面主要是对存储容量、过期时间、缓存等做一些设置。常用设置信息如下：

```
import { AsyncStorage } from 'react-native';
var storage = new Storage({
  // 最大容量，默认值 1000 条数据循环存储
  size: 1000,
  // 存储引擎：对于 RN 使用 AsyncStorage，对于 web 使用 window.localStorage
  // 如果不指定则数据只会保存在内存中，重启后即丢失
  storageBackend: AsyncStorage,
  // 数据过期时间，默认一整天，设为 null 则永不过期
  defaultExpires: 1000 * 3600 * 24,

  // 读写时在内存中缓存数据。默认启用。
  enableCache: true,

  // 如果 storage 中没有相应数据，或数据已过期，
  // 则会调用相应的 sync 方法，无缝返回最新数据。
  sync: require('./sync') // sync 文件是要你自己写的
});
```

数据的保存、读取和删除操作

接下来，我们就可以使用 Storage 的实例对数据进行存取、读取和删除等操作了。相关示例代码如下：

```
// 使用 key 来保存数据
storage.save({
  key: 'loginState',
  data: {
    from: 'some other site',
    userid: 'some userid',
    token: 'some token'
  },
  expires: 1000 * 3600
});
// 读取数据
```

```

storage.load({
  key: 'loginState',

  // autoSync(默认为true)意味着在没有找到数据或数据过期时自动调用相应
  // 的 sync 方法
  autoSync: true,
  syncInBackground: true,
  syncParams: {
    extraFetchOptions: {
      // 各种参数
    },
    someFlag: true,
  },
}).then(ret => {
  // 如果找到数据,则在 then 方法中返回
  this.setState({ user: ret });
}).catch(err => {
  // 如果没有找到数据且没有 sync 方法,或者有其他异常,则在 catch 中返回
  switch (err.name) {
    case 'NotFoundError':
      // TODO;
      break;
    case 'ExpiredError':
      // TODO
      break;
  }
})

```

在本天气预报项目中,为了实现数据的持久化,只需要将接口返回的数据转换为字符串存储起来即可。当在无网环境下,系统会默认读取本地的持久化数据。使用 react-native-storage 库持久化数据的相关代码如下:

```

// 保存接口返回数据到 react-native-storage 中
saveLocalCityData() {
  storage.save({
    key: 'cities',
    data: JSON.stringify(this.cityList)
  })
}

```

在无网环境下系统读取本地存储的数据来绘制到界面上,代码如下:

```

// 读取本地持久化数据
loadLocalCityData() {
  storage.load({
    key: 'cities',
    // autoSync(默认为true)

```

```

autoSync: true,
// syncInBackground(默认为true)意味着如果数据过期,
// 在调用 sync 方法的同时先返回已经过期的数据。
// 设置为 false 的话,则始终强制返回 sync 方法提供的最新数据(当然会需要更多等待时间)。
syncInBackground: true,
}).then(ret => {
  let array = JSON.parse(ret);
  for (let i = 0; i < array.length; i++) {
    this.cityList.push(array[i]);
  }
  this.cityList = this.removeDuplicatedItem(this.cityList);
}).catch(err => {
  // 如果没有找到数据且没有 sync 方法,
  // 或者有其他异常,则在 catch 中返回
  console.warn(err.message);
  switch (err.name) {
    case 'NotFoundError':
      alert('读取失败')
      break;
    case 'ExpiredError':
      alert('读取失败')
      break;
  }
});
}

removeDuplicatedItem(ar) {
  var ret = [];
  ar.forEach(function (e, i, ar) {
    if (ar.indexOf(e) === i) {
      ret.push(e);
    }
  });
  return ret;
}

```

10.3.4 工具类

俗话说,工欲善其事,必先利其器。为了方便开发,往往需要将一些和逻辑无关的工具类代码,单独放到一个工具文件包中这个工具包常常封装一些诸如尺寸转换、网络、时间、日期等工具类。在本项目中,我们主要是封装一个对时间进行操作的工具类。

```
// 时间工具类
```

```
/**
```

```
* 根据日期获取星期
```

```

    * @param date
    * @returns {*}
    */
function getWeekdayByDate(date) {
    let a = new Array("周日", "周一", "周二", "周三", "周四", "周五", "周六");
    let day = new Date(date).getDay();
    var today = getNowFormatDate();
    var tomorrow = getTomorrowFormatDate();
    if (today === date)
        return '今天';
    else if (date === tomorrow)
        return '明天';
    else
        return a[day];
}

/**
 * 获取今天的时间戳
 * @returns {string}
 */
function getNowFormatDate() {
    var date = new Date();
    var seperator1 = "-";
    var month = date.getMonth() + 1;
    var strDate = date.getDate();
    if (month >= 1 && month <= 9) {
        month = "0" + month;
    }
    if (strDate >= 0 && strDate <= 9) {
        strDate = "0" + strDate;
    }
    var currentdate = date.getFullYear() + seperator1 + month + seperator1 +
strDate
    return currentdate;
}

/**
 * 获取明天的时间戳
 * @returns {string}
 */
function getTomorrowFormatDate() {
    var date = new Date();
    date.setTime(date.getTime() + 24 * 60 * 60 * 1000);
    var seperator1 = "-";
    var month = date.getMonth() + 1;
    var strDate = date.getDate();
    if (month >= 1 && month <= 9) {

```

```

        month = "0" + month;
    }
    if (strDate >= 0 && strDate <= 9) {
        strDate = "0" + strDate;
    }
    var currentdate = date.getFullYear() + seperator1 + month + seperator1 + strDate
    return currentdate;
}

/**
 * 根据时间戳获取月日
 * @param date
 */
function getMonthAndDayByDate(date) {
    return date.substring(8,date.length)+' 日 ';
}

/**
 * 根据时间戳获取时分
 * @param date
 */
function getHoursAndMinsByDate(date) {
    return date.substring(11,date.length);
}

// 导出封装的方法
module.exports = {
    getWeekdayByDate: getWeekdayByDate,
    getMonthAndDayByDate: getMonthAndDayByDate,
    getHoursAndMinsByDate: getHoursAndMinsByDate
}

```

当需要调用上面封装的方法的时候，先导入 DateUtil.js 文件，然后调用相关的方法即可。

```

import dateUtil from '../util/DateUtil';
//...
<Text style={styles.text}>{dateUtil.getHoursAndMinsByDate(data.date)}</Text>

```

10.4 模块开发

模块化，是指解决一个复杂问题时，自上向下逐层把软件系统划分成若干模块的过程，是软件解决复杂问题的重要手段。每个模块完成一个特定的子功能，所有的模块按某种方法组装起来，成为一个整体，完成系统设计所要求的功能。

采用模块化开发，能在一定程度上降低程序的耦合度和复杂度。在本示例项目中，我们将项目按功能分为天气页面展示模块和选择城市模块。

10.4.1 组件封装

在前端页面开发中,为了让页面逻辑更加清晰化,我们建议将一些具有相同功能的代码抽取出来封装成公共的组件。常见的如分割线、弹窗提示等。

```

Divider.js
'use strict';
import React, {Component} from 'react'
import {observer} from 'mobx-react'
import {StyleSheet, View} from 'react-native';

@observer
export default class Divider extends Component {

  constructor(props) {
    super(props);
    this.state = {};
  }

  render() {
    let dividerHeight = this.props.dividerHeight;
    if (dividerHeight===null)
      dividerHeight=1;
    return (
      <View style={[styles.container,{height:dividerHeight}]}>
        </View>
    )
  }
}

const styles = {
  container: {
    flex: 1,
    backgroundColor: 'rgba(100,100,100,0.1)'
  },
},

```

除了对公共的代码进行封装处理之外,在项目开发中,对于复杂的页面也需要拆分出来单独封装成组件,通过将一些复杂的页面拆分为若干自定义的组件,从而减少代码复杂度,增强代码的可阅读性和可维护性。

10.4.2 天气预报页面开发

在 React Native 开发中,对于比较复杂的页面,我们建议对界面进行拆解,然后封装成可以单独使用的小模块,这样不仅减少开发的难度,也方便其他开发人员对代码的维护。

在本项目中，根据返回数据和页面展示情况，先将天气首页分为 5 个子模块进行处理，分别为：当前天气信息、当天天气时段信息、未来 7 天天气列表、空气指数、生活指数建议。对应代码如下：

```
// 当前天气数据信息 WeatherIndex.js
'use strict';
import React, {Component} from 'react'
import {StyleSheet, View, Image, ScrollView, RefreshControl} from 'react-native';
import Header from '../components/HeaderContent'
import DailyForecast from '../components/DailyForecast'
import {observer} from 'mobx-react/native'
import weatherStore from '../stores/WeatherStore'
import stateStore from '../stores/StateStore'
import HourlyForecast from '../components/HourlyForecast'
import ApiConfig from '../config/Index'
import Divider from '../components/Divider'
import AirCondition from '../components/AirCondition'
import LifeSuggestion from '../components/LifeSuggestion'

@observer
export default class WeatherIndex extends Component {

  constructor(props) {
    super(props);
  }

  componentWillMount() {
    this._refreshWeatherData();
    stateStore.loadLocalCityData();
  }

  _refreshWeatherData = () => {
    weatherStore.requestWeatherByName(weatherStore.currentCityName);
  }

  _handleScrollEvent = (event) => {
    let offsetY = event.nativeEvent.contentOffset.y;
    if (offsetY > 250) {
      stateStore.scrollToEnd = true
    } else {
      stateStore.scrollToEnd = false
    }
  }
}
```

```

// 根据环境变量绘制界面
render() {
  if (__ANDROID__) {
    return this._renderAndroid();
  } else {
    return this._renderIOS();
  }
}

// 渲染 Android
_renderAndroid = () => {
  return (
    <View style={styles.container}>
      <Image style={styles.image} source={{uri: ApiConfig.
backgroundWallpaper}}
        resizeMode={'scale'} blurRadius={25}>
      <ScrollView style={styles.container}
        scrollEventThrottle={200}
        onScroll={(e) => this._handleScrollEvent(e)}
        showsVerticalScrollIndicator={false}
        refreshControl={
          <RefreshControl
            refreshing={weatherStore.loading}
            onRefresh={this._refreshWeatherData}
            tintColors={'white'}
            titleColors={'white'}
            title={weatherStore.loading ? "刷新
中....." : '下拉刷新 '}/>
        <Header/>
        <Divider/>
        <HourlyForecast/>
        <Divider/>
        <DailyForecast/>
        <AirCondition/>
        <LifeSuggestion/>
      </ScrollView>
    </Image>
  </View>
)
}

// 渲染 IOS
_renderIOS = () => {
  return (
    <View style={styles.container}>
      <Image style={styles.container} source={{url: ApiConfig.
backgroundWallpaper}}

```

```

        resizeMode={'scale'} blurRadius={25}>>
        <ScrollView style={styles.container}
          scrollEventThrottle={200}
          onScroll={(e) => this._handleScrollEvent(e)}
          showsVerticalScrollIndicator={false}
          refreshControl={
            <RefreshControl refreshing={weatherStore.loading}
              onRefresh={this._refresh Weather
Data)

          tint color={'white'}
          title color={'white'}
          title={weatherStore.loading ? "刷新中……" : '下拉刷新'}/>>
          <Header navigation={this.props.navigation}/>
          <Divider/>
          <HourlyForecast/>
          <Divider/>
          <DailyForecast/>
          <AirCondition/>
          <LifeSuggestion/>
        </ScrollView>
      </Image>
    </View>
  )
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  image: {
    flex: 1,
  },
  addImage: {
    height: 45,
    width: 45,
  },
  text: {
    fontSize: 18,
    color: '#999999',
    marginTop: 20,
    textAlign: 'center'
  }
});

```

在上面的代码中，我们将整个页面拆分成若干小页面，并将小页面封装成一个可以单独使

用的组件，然后将封装的组件按照某种布局样式组装到主页面，从而完成主页面的绘制工作。

例如，下面对空气质量指数的封装，此页面从上到下分为：标题、AQI 指数、详细参数。在 AQI 指示点的实现上，通过 Text 标签将横条分为 6 段，然后通过区间条件的判断来确定它的显示位置。最后，通过 Mobx 数据监听来实现页面的刷新操作。本视图部分的效果如图 10-6 所示。

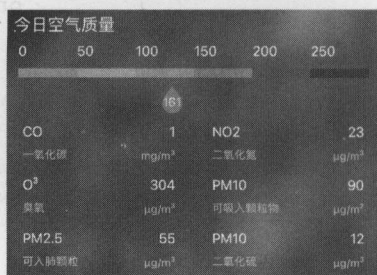


图10-6 空气质量指数

本部分代码主要涉及两个文件：AirCondition.js 和 AirAqiItem.js。其中，AirCondition.js 是主页面，AirAqiItem.js 是 AQI 网格数据界面。

```
// 空气质量 AirCondition.js
'use strict';
import React, {Component} from 'react';
import {StyleSheet, View, Text, Image} from 'react-native';
import {observer} from 'mobx-react/native';
import weatherStore from '../stores/WeatherStore';
import Divider from './Divider'
import AqiItem from './AirAqiItem'
```

```
@observer
```

```
export default class AirCondition extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = {};
  }
```

```
  _handleIndicatorColor = (value) => {
    if (value < 50) {
      return airColors[0];
    } else if (50 < value && value < 100) {
      return airColors[1];
    } else if (100 < value && value < 150) {
      return airColors[2];
    }
  }
```

```

    else if (150 < value && value < 200) {
        return airColors[3];
    }
    else if (200 < value && value < 250) {
        return airColors[4];
    } else {
        return airColors[5];
    }
}

render() {
    let marginLeftValue = 0;
    if (!weatherStore.loading) {
        let weatherData = weatherStore.getCurrentCityWeather();
        if (weatherData !== null) {
            marginLeftValue = weatherData.aqi.city.aqi;
        }
    }
    let offset = marginLeftValue;
    if (offset > 330)
        offset = 330;
    return (
        <View style={styles.container}>
            <Divider dividerHeight={20}/>
            <Text style={[styles.text, {marginLeft:10,fontSize:18,margint
op:10}]}>今日空气质量</Text>
            <View style={styles.indicatorContainer}>
                <Text style={styles.text}>0</Text>
                <Text style={styles.text}>50</Text>
                <Text style={styles.text}>100</Text>
                <Text style={styles.text}>150</Text>
                <Text style={styles.text}>200</Text>
                <Text style={styles.text}>250</Text>
            </View>
            <View style={styles.indicatorContainer}>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[0]}]}/>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[1]}]}/>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[2]}]}/>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[3]}]}/>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[4]}]}/>
                <View style={[styles.indicator,{backgroundColor:airColo
rs[5]}]}/>
            </View>
        </View>
    )
}

```

```

</View>
<View style={styles.currentIndicator}>
  <Image source={require('../images/water.png')}
    style={[styles.indicatorImage, {tintColor: this._handleIn
indicatorColor (marginLeftValue), marginLeft: (parseInt (offset)+5) }]}>
    <Text style={styles.indicatorText}>{marginLeftVal
ue}</Text>
  </Image>
</View>
<View style={styles.detailColumnContainer}>
  <View style={styles.detailRowContainer}>
    <AqiItem index={0}/>
    <AqiItem index={1}/>
  </View>
  <View style={styles.detailRowContainer}>
    <AqiItem index={2}/>
    <AqiItem index={3}/>
  </View>
  <View style={styles.detailRowContainer}>
    <AqiItem index={4}/>
    <AqiItem index={5}/>
  </View>
</View>
<Divider dividerHeight={20}/>
</View>
)
}
}

const airColors = ['#73BB4D', '#EBB541', '#FC9B56', '#F17751', '#A94057', '#7B1F3C'];

const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 20
  },
  indicatorContainer: {
    flex: 1,
    flexDirection: 'row',
    alignItems: 'center',
    marginLeft: 15,
    marginRight: 15,
    marginTop: 10
  },
  indicator: {
    flex: 1,
    height: 10
  }
});

```

```

    },
    currentIndicator: {
      height: 10
    },
    text: {
      fontSize: 15,
      color: 'white',
      flex: 1,
      backgroundColor: 'transparent'
    },
    indicatorImage: {
      width: 35,
      height: 35,
      justifyContent: 'center',
      alignItems: 'center',
      marginTop: 5
    },
    indicatorText: {
      fontSize: 13,
      color: 'white',
      backgroundColor: 'transparent',
      marginTop: 8
    },
    detailColumnContainer: {
      flexDirection: 'column',
      marginTop: 30
    },
    detailRowContainer: {
      flexDirection: 'row'
    }
  });
}

AirAqiItem.js
'use strict';
import React, {Component} from 'react';
import {StyleSheet, View, Text, ActivityIndicator} from 'react-native';
import {observer} from 'mobx-react/native';
import weatherStore from '../stores/WeatherStore';

@observer
export default class AqiItem extends Component {

  constructor(props) {
    super(props);
    this.state = {};
  }

```



```

render() {
  let itemIndex=this.props.index;
  if (weatherStore.loading) {
    return this._renderLoading();
  } else {
    return this._renderContent(weatherStore.aqiList[itemIndex]);
  }
}

_renderLoading = () => {
  return (
    <View style={styles.container}>
      <ActivityIndicator/>
    </View>
  )
}

_renderContent = (item) => {
  return (
    <View style={styles.container}>
      <View style={styles.columnItem}>
        <Text style={styles.textTop}>{item.eng_name}</Text>
        <Text style={styles.textTop}>{item.value}</Text>
      </View>
      <View style={styles.columnItem}>
        <Text style={styles.textBottom}>{item.chn_name}</Text>
        <Text style={styles.textBottom}>{item.unit}</Text>
      </View>
      <View style={styles.divider}>
      </View>
    </View>
  )
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    flexDirection: 'column',
    backgroundColor: 'transparent',
  },
  columnItem: {
    flex: 1,
    flexDirection: 'row',
    justifyContent: 'space-between',
    marginLeft: 20,
    marginRight: 20,
    marginTop: 10,
  },
  textTop: {

```

```

    fontSize:15,
    color: 'white'
  },
  textBottom:{
    color:'rgb(211,211,211)',
    fontSize:12
  },
  divider: {
    backgroundColor: 'rgba(100,100,100,0.2)',
    marginLeft: 10,
    marginRight: 10,
    marginTop: 5,
    height: 1
  }
});

```

在使用 React Native 开发复杂页面时，往往需要对页面按照实际情况进行拆分，这与移动开发的思路是一样的，也是为了减少代码的复杂度。关于城市列表的实现逻辑，读者可以自行查看随书源码。

10.4.3 Navigation导航

一个完整的移动应用项目包含若干的页面，而页面与页面之间的联系往往是通过导航来实现的。React Native 默认为我们提供了导航组件 Navigator。在本项目中，我们使用更加灵活的第三方库 react-navigation。关于它的介绍和使用我们前面已经讲过，这里不再详述，读者可以翻看前面章节的介绍。

不过需要注意的是，在使用 react-navigation 进行导航的过程中，由于我们使用了分块开发的思路，所以在子模块中如果需要使用 Navigation 时，需要将 Navigation 传入到子组件中，不然会报如下的错误，具体如图 10-7 所示。

```
undefined is not an object (evaluating 'navigation.navigate')
```

```
undefined is not an object (evaluating
'navigation.navigate')
```

```
onPress
```

```
index.android.bundle?
platform=android&dev=true&hot=false&minify=false:
55452:54
```

```
touchableHandlePress
```

```
index.android.bundle?
platform=android&dev=true&hot=false&minify=false:
29666:59
```

图10-7 Navigation报错

10.5 运行结果

至此，一款基于 LBS 的简单天气预报移动应用就开发完成了，虽然页面比较简单，但基本具备了移动应用所要具备的基本功能，本例代码已随书提供，运行效果如图 10-8 所示。

功能开发完成后，如果想要发布到应用商店上供大家下载，读者只需要按照前面的介绍制作签名包，然后上传到应用市场即可。



图10-8 运行效果图

第

11

章

O2O移动团购应用

当今从全球范围来看，传统的零售行业正在经历着产业的变革和新一轮的产业升级，在国内亦是如此，特别是有不少曾经创造过辉煌的传统零售行业，面对互联网技术和新兴技术的迅猛发展，也只能勉强维持生存。在这样的背景下，O2O成为商家解决当前困局的最广泛的商业模式。

O2O（全称 Online To OffLine），是指线上营销购买带动线下消费的一种营销模式，这个概念最早由美国支付公司 TrialPay 创始人 Alex Rampell 在 2010 年 8 月提出，而后被引入中国，并搭上了团购市场的顺风车，迅速在生活服务领域开花结果。

如今，O2O 成为大众所熟知的商业模式。O2O 商业模式的本质是将线下的商务机会与互联网结合，让互联网成为线下交易的前台，通过互联网与线下的完美对接，让消费者在享受线上优惠价格的同时，又可享受线下贴身的服务。O2O 的概念从 2010 年被提出来之后，先后经历了 3 个阶段的发展。

在早期的 1.0 版本中，O2O 主要是完成线上线下初步对接的工作，利用线上推广把相关的用户集中起来，然后把线上的流量导到线下，主要领域集中在以美团为代表的线上团购和促销等领域。这一阶段，平台和用户的互动较少，基本上以交易的完成为终结点。用户更多是受价格等因素驱动，购买和消费频率等也相对较低。

发展到 2.0 阶段后，O2O 基本为消费者所理解和接受。这个阶段最主要的特色就是服务性电商模式：包括商品（服务）、下单、支付等流程，把之前的电商模式转移到生活化场景中来，

从而完成线上交易。在这一阶段,涌现出了上门按摩、上门送餐、上门生鲜、上门化妆、滴滴打车等各种 O2O 模式。

到了 3.0 阶段, O2O 的概念更加细化, 垂直细分领域的优势开始凸显出来。而另一方面, 垂直细分领域通过自己专业化的技术开始向平台化模式发展, 由原来的细分领域解决某个痛点的模式开始横向扩张, 覆盖到整个行业。

11.1 需求分析

根据易观智库提供的最新数据来看, 团购类服务主流消费人群为 30 岁以下的中低收入者, 以女性用户为主。从区域分布来看, 主要分布于一二线城市, 其中收入在 4999 元以下的用户占比为 %85.2, 女性用占比为 64.2%。团购业务主要集中在一二线省会城市, 年轻人为消费团购业务消费主体, 这是由于一二线城市互联网环境相对来说要发达很多, 互联网氛围经过多年的发展已经成熟。随着智能手机的普及以及基础网络建设的升级, 用户的互联网使用习惯已经被慢慢培养起来。年轻人从小就接触互联网, 对新事物比较好奇、愿意尝试、接受学习能力也更强, 是互联网的主流消费人群。

11.1.1 需求分析

通过对团购目标用户人群划分可知, 团购类移动应用的主流人群应为: 30 岁以下以女性为主的中低收入者, 这部分用户接触互联网的时间较长, 对互联网产品的认知度比较高, 且对价格相对敏感, 同时追求一定的生活品质。所以对这部分主流用户而言, 需要找到一个能够提供“物美价廉”商品的互联网平台, 让其能够以最小的成本获取较优质的生活服务。

作为团购类产品的入口, 餐饮美食以及娱乐性需求占人们日常生活消费中的很大比例。团购平台通过提供大量团购优惠服务来满足用户的需求, 在此基础上不断扩展其他方面的团购优惠服务, 从而全方位地满足人们在日常生活中的需求。

而对于线下实体店铺的商家来说, 加入相对廉价的团购平台, 不仅满足了商家的宣传需求, 扩大了知名度, 还能够增加自己的销售额。综上可知, 团购平台同时满足了用户以低廉的价格获取较优质服务的需求以及商家增加销售额、增加知名度的需求。

随着互联网经济的到来, 电子商务模式渗透到贸易活动的各个阶段, 包括信息交换、售前售后服务、销售、电子支付、运输、组建虚拟企业、共享资源等。电子商务的参与者包括消费者、销售商、供货商、企业雇员, 等等。而电子商务的目的是要实现企业乃至全社会的高效率、低成本的贸易活动。

根据产品的定位和对产品目标用户的分析, 用户使用团购类产品的场景大体可以分为以下几类。

- 有消费计划且目标明确: 根据调查机构提供的数据, 有明确消费计划并且知道消费类别的用户为团购类 APP 的主用户。对于这类用户, 操作流程一般分为以下两种: 第一种是通过团购类产品提供的商品分类, 直接对商家的团购优惠商品进行消费; 第二种是在用

户没有找到清晰的分类入口或者商家的情况下，通过搜索、筛选等操作进行团购消费。

所以基于这种情况，团购类产品都会在首页配置大量的分类入口，并在醒目的位置内置搜索框，从而方便用户找到想要的团购产品。

- 有消费计划单没有明确目标：在有消费计划但是没有明确的消费目标的场景下，用户更多地会依赖APP展示的分类、相关的运营活动以及基于地理位置推荐的商家列表来进行团购消费。在此场景下，用户主要通过分类及筛选找到自己满意的团购信息，或者通过查看附近的商家来找到理想的商家。所以，在这种场景下，基于LBS的商家推荐服务显得很有必要。
- 进店消费，网上结账：用户在某商家消费完毕后，使用团购APP结账，这类消费的主要流程为：利用“扫一扫”扫描商家的二维码付款，或者给商家出示自己的付款码，商家扫描完成结账。在这种场景下，用户不仅能够便捷地完成付款操作，还能享受消费上的折扣优惠。
- 商家活动或者平台活动，消费优惠：在团购产品中，为了吸引用户，平台会联合部分商家定时推出一些专题活动。在这类场景中，用户能够很快地进入相关活动主页了解优惠信息，从而进行消费。
- 其他场景：在很多情况下，用户对于商家的选择都是漫无目的的。哪里人多，哪里的口碑好，就在哪里消费。针对这种场景。位置、评分、网友的评价是用户的重要标准。其次，对于平台来说，如果在某个商家消费后能够获取积分等，也能激发用户的消费欲望。

11.1.2 功能分析

通过参考市面上主流的团购类移动应用可知，团购类产品本身的流程、功能、界面非常相似，用户群也较为固定，所以团购类产品的竞争更多的是争夺商家团购资源以及有效地展开运营，通过覆盖更多的城市，涵盖更多的商家优惠以及提供更加优质的服务来打动消费者。同时大力去扶植发展比较迅速的垂直细分领域，力争在本地生活市场占据更大的份额，涵盖生活服务中的方方面面。

本团购示例项目参考了市面上部分团购类应用，但并不提供完整的应用功能。通过对主流团购应用的参考可知，一个典型团购应用的主要页面往往包含：由首页、附近、订单、个人中心组成的主页面，产品分类页面，产品详情页面，购买页面，账号系统等。

在本示例项目中，主要包含以下几个模块：首页、附近、订单、个人中心。其中，首页最为重要，如何设计首页，是产品设计中一门很高深的学问。首页模块，主要提供团购分类导航入口、固定的专题运营活动、限时抢购产品和推荐产品等。附近模块，主要提供基于LBS的附近商家及分类商家信息以及提供商户搜索历史和标签。订单，主要用于展示订单信息，订单模块可以查看个人订单相关的信息。个人中心，主要提供个人信息的维护和二级页面的跳转。同时，在本项目，当用户点击分类导航入口的时候，还可以跳转到商品分类页面，点击某个商品或服务时还可以跳转到页面详情。在实际使用过程中，为了方便用户访问H5活动页面，需要

对 WebView 进行封装。

综合来说,对于本示例项目,并不需要太复杂的实现。希望通过学习本示例项目,读者可以学习到从项目搭建到上线的完整过程,并对前面章节介绍的内容进行综合应用。在学习本项目之前,先预览下项目的部分效果,如图 11-1 所示。



图11-1 团购应用部分效果

11.2 应用设计

当前,团购类网站主要向着大而全的平台化方向发展,通过商家入驻、平台管理的模式,团购类网站基本覆盖了人们生活中最基本的消费。以美团网和口碑网为例,用户可以在美团网订酒店、买电影票、购物、购买旅游产品等,然后使用线上支付环境直接进行线上支付。

“互联网+”产品设计催生了O2O产品设计,相对于传统的工业产品设计,O2O具有先天的优势,并积极影响着O2O产品的进一步发展。在产品设计的进程中,研究分析产品的未来发展趋势具有非常重要的意义。

通过前面对产品的开发背景和需求进行分析,可以很清晰地发现:O2O产品的设计就是将产品设计放到生活消费领域中,通过互联网将虚拟世界和现实世界相结合,形成的一种消费模式。在该模式中,营销、消费、交易体验等模块是其最基本的组成部分。

在本系统中,并不会涉及太复杂的系统及理论,只需要根据需求分析阶段确定的软件的功能,确定软件系统的整体结构、功能模块、每个模块的算法实现即可。软件设计的最终目的是帮助开发者在开发之前形成具体的开发方案。在本系统中,模拟实际生活中的交易流程如图11-2所示。

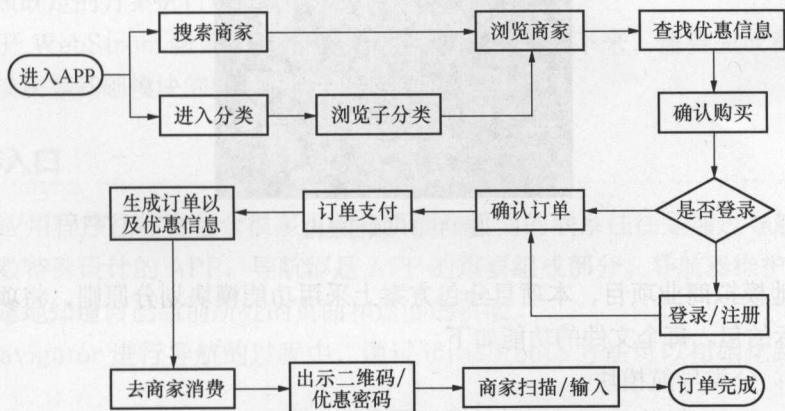


图11-2 团购类APP用户主流程

11.2.1 模块划分

在软件项目开发中,为了方便代码的管理和维护,通常是将同类业务放到一个文件夹中。这种模块化的开发思路,可以有效减少循环依赖、代码间的耦合,以此提高软件的开发效率。同时模块化的开发思路也方便对团队成员进行管理。为了说明本团购项目的模块划分思路,首先来看一下本项目的结构,如图11-3所示。

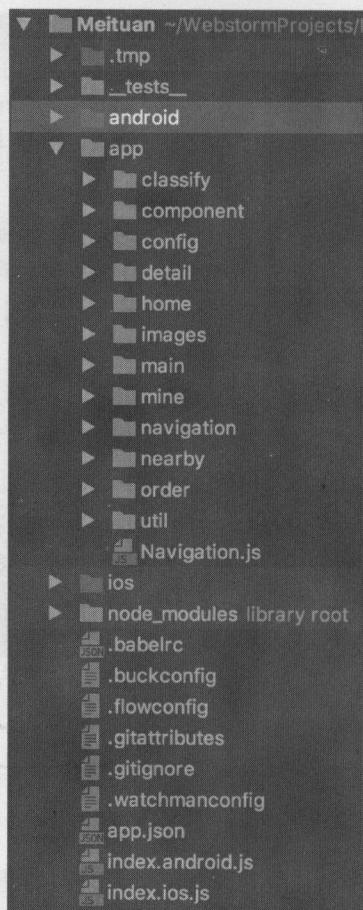


图11-3 项目结构

为了真实地模拟商业项目，本项目分包方案上采用功能模块划分原则，将项目安装功能模块分为上图所示的包。每个文件的功能如下。

- `classif`: 分类导航模块。
- `component`: 封装的自定义组件或公共组件。
- `config`: 项目的配置文件（模拟接口、数据等）。
- `detail`: 团购详情页及相关二级页面。
- `home`: 主页面首页模块。
- `images`: 本项目图片资源文件。
- `util`: 工具类文件。

11.2.2 添加第三方库

在软件开发过程中，为了达到快速开发的目的，常常需要引入一些性能相对高效，且使用

便捷的第三方库。在本示例项目中，引入的第三方库如下：

```
"native-base": "^2.3.0",
"react-native-blur": "^3.1.2",
"react-native-deprecated-custom-components": "^0.1.1",
"react-native-parabolic": "^1.1.1",
"react-native-scrollable-tab-view": "^0.6.7",
"react-native-tab-navigator": "^0.3.3",
"react-native-vector-icons": "^4.2.0",
```

其中，这些第三方库提供的主要功能描述如下。

- react-native-blur：提供毛玻璃效果。
- react-native-scrollable-tab-view：提供Tab切换效果。
- react-native-vector-icons：提供提供常用的视图图标。

合理使用第三方库，不但可以提升项目的开发效率，还可以解决产品在持续迭代过程中出现的性能问题。

11.3 项目搭建与工具模块开发

通过前面对功能的理解和对模块的划分，现在我们对项目的整体情况有了大体的了解，接下来只需要按照既定的方案进行编码即可。

首先，打开 WebStrom 新建一个 React Native 项目，接下来，需要完成程序入口、基本模块、工具类模块等基础模块的编写。

11.3.1 程序入口

一个移动应用程序往往会包含很多页面，页面与页面的联系往往是通过导航器跳转来实现的。除了极少数特殊设计的 APP，导航都是 APP 的重要组成部分。导航器维护一个导航堆栈，能够让用户清楚地知道自己当前所处的页面和返回的页面。

在使用 Navigator 进行导航的过程中，通过 `initialRoute` 方法可以初始化路由，其他相关配置如下：

```
<Navigator
  initialRoute={{component: MainScreen }}
  configureScene={() => Navigator.SceneConfigs.PushFromRight}
  renderScene={(route, navigator) => {
    return <route.component navigator={navigator} {...route.args}/>
  }}
/>
```

为了区分 Android 和 iOS 平台的一些显示效果，需要对平台进行判断，然后绘制不同的显示效果。而 Navigator 在 0.44 版本之后是默认过期的，如果你在 0.44 版本以后继续使用 Navigator，需要导入相关的依赖包。Navigation.js 文件完整代码如下：

```

import React, { Component } from 'react'
import {View, StatusBar, Platform } from 'react-native'
import {Navigator } from 'react-native-deprecated-custom-components'
import MainScreen from '../main/MainScreen'

export default class Navigation extends Component{

  render(){
    return Platform.OS == "ios"? (
      <Navigator
        initialRoute={{component: MainScreen}}
        configureScene={() => Navigator.SceneConfigs.FloatFromRight}
        renderScene={(route, navigator) => {
          return <route.component navigator={navigator}
            {...route.args}/>
        }}
      </>
    ):(
      <View style={{flex: 1}}>
        <StatusBar
          backgroundColor="#0398ff"
          barStyle="light-content"/>
        <Navigator
          initialRoute={{component: MainScreen}}
          configureScene={() => Navigator.SceneConfigs.PushFromLeft}
          renderScene={(route, navigator) => {
            return <route.component navigator={navigator}
              {...route.args}/>
          }}
        </>
      </View>
    )
  }
}

```

在早期版本中，官方推荐使用 Navigator 进行页面导航。不过在 0.44 版本之后，由于 Navigator 性能较差，在 iOS 应用上，官方推荐使用 NavigatorIOS，而 Android 平台不得不继续使用 Navigator。对于系统已经废弃的组件，如果想要再次使用，需要使用如下的方式导入：

```

// 导入 Navigator
import {Navigator } from 'react-native-deprecated-custom-components'

```

11.3.2 搭建主框架

在移动应用开发过程中，使用 TabBar 样式进行 Tab 页面的切换是移动应用设计中比较经典的主页面导航方式。在本项目中，主页面由 5 个 Tab 页面组成，当点击某个 Tab 后会跳转相

应的视图，其效果如图 11-4 所示。

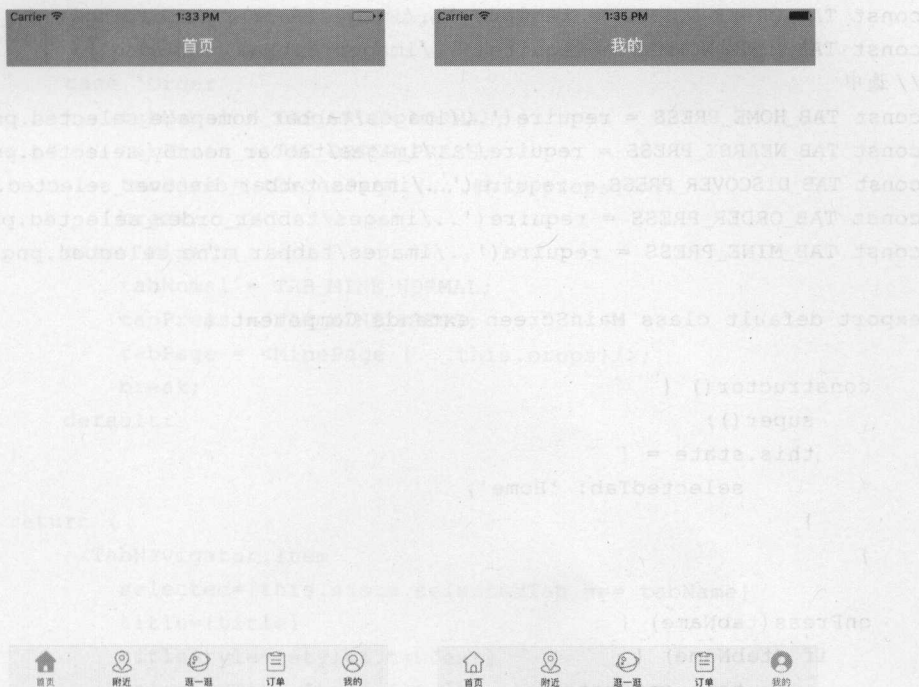


图11-4 Tab导航方式

在 React Native 开发中，要实现 Tab 切换导航效果，最简单的方式莫过于使用第三方库：react-native-tab-navigator，使用 react-native-tab-navigator 库可以很轻松地实现 Tab 切换。MainScreen.js 主页面代码如下：

```
import React, {Component} from 'react';
import {
  StyleSheet,
  Image,
  View
} from 'react-native';
import TabNavigator from 'react-native-tab-navigator';

import HomePage from './HomePage';
import NearbyPage from './NearbyPage';
import DiscoverPage from './DiscoverPage';
import OrderPage from './OrderPage';
import MinePage from './MinePage';

// 默认选项
const TAB_HOME_NORMAL = require('../images/tabbar_homepage.png');
const TAB_NEARBY_NORMAL = require('../images/tabbar_nearby.png');
```



```

const TAB_DISCOVER_NORMAL = require('../images/tabbar_discover.png');
const TAB_ORDER_NORMAL = require('../images/tabbar_order.png');
const TAB_MINE_NORMAL = require('../images/tabbar_mine.png');
// 选中
const TAB_HOME_PRESS = require('../images/tabbar_homepage_selected.png');
const TAB_NEARBY_PRESS = require('../images/tabbar_nearby_selected.png');
const TAB_DISCOVER_PRESS = require('../images/tabbar_discover_selected.png');
const TAB_ORDER_PRESS = require('../images/tabbar_order_selected.png');
const TAB_MINE_PRESS = require('../images/tabbar_mine_selected.png');

```

```

export default class MainScreen extends Component {

```

```

  constructor() {
    super();
    this.state = {
      selectedTab: 'Home',
    }
  }

```

```

  onPress(tabName) {
    if (tabName) {
      this.setState({
        selectedTab: tabName,
      });
    }
  }

```

```

  renderTabView(title, tabName, defaultTab, isBadge) {
    var tabNormal;
    var tabPress;
    var tabPage;
    switch (tabName) {
      case 'Home':
        tabNormal = TAB_HOME_NORMAL;
        tabPress = TAB_HOME_PRESS;
        tabPage = <HomePage {...this.props}/>;
        break;
      case 'Nearby':
        tabNormal = TAB_NEARBY_NORMAL;
        tabPress = TAB_NEARBY_PRESS;
        tabPage = <NearbyPage/>;
        break;
      case 'Discover':
        tabNormal = TAB_DISCOVER_NORMAL;

```

```

        tabPress = TAB_DISCOVER_PRESS;
        tabPage = <DiscoverPage/>;
        break;
    case 'Order':
        tabNomal = TAB_ORDER_NORMAL;
        tabPress = TAB_ORDER_PRESS;
        tabPage = <OrderPage {...this.props}/>;
        break;
    case 'Mine':
        tabNomal = TAB_MINE_NORMAL;
        tabPress = TAB_MINE_PRESS;
        tabPage = <MinePage {...this.props}/>;
        break;
    default:
}

return (
    <TabNavigator.Item
        selected={this.state.selectedTab === tabName}
        title={title}
        titleStyle={styles.tabText}
        selectedTitleStyle={styles.selectedTabText}
        renderIcon={() => <Image style={styles.icon} source=
{tabNomal}/>}
        renderSelectedIcon={() => <Image style={styles.icon} source=
{tabPress}/>}
        onPress={() => this.onPress(tabName)}>

        <View style={styles.page}>
            {tabPage}
        </View>
    </TabNavigator.Item>
);

},

render() {
    return (
        <View style={styles.container}>
            <TabNavigator
                tabBarStyle={styles.tabStyle}>
                {this.renderTabView('首页', 'Home', HomePage, false)}
                {this.renderTabView('附近', 'Nearby', NearbyPage, false)}
                {this.renderTabView('逛一逛', 'Discover', DiscoverPage,
false)}
                {this.renderTabView('订单', 'Order', OrderPage, false)}
            </TabNavigator>
        </View>
    );
}

```

```

        {this.renderTabView('我的', 'Mine', MinePage, false)}
      </TabNavigator>
    </View>
  );
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1
  },
  tabStyle: {
    alignItems: 'center',
    justifyContent: 'center',
  },
  tabText: {
    fontSize: 10,
    color: 'black'
  },
  selectedTabText: {
    fontSize: 10,
    color: 'green'
  },
  icon: {
    width: 25,
    height: 25,
    alignItems: 'center',
    justifyContent: 'center',
  },
  page: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFFFFF'
  }
});

```

在上面的代码中，为了方便使用，我们封装了一个 `renderTabView()`，在此方法中，通过加入 `switch` 条件判断来确定当前子页面（`HomePage`、`NearbyPage` 为 `Tab` 的子页面）是否被选中。当子页面被选中之后即为当前页面，在 `TabNavigator.Item` 中使用 `selected` 将被选中的页面设置为被选择的页面即可。

11.3.3 导航栏封装

在移动应用程序中，导航栏是最基本的组成部分，其样式也是各不相同。一个通用的导航

栏左边是返回按钮，居中显示标题，靠右还有分享按钮等，其效果大体如图 11-5 所示。



图11-5 通用头部导航栏

在封装这类导航栏的过程中，为了满足多种场景的使用需求（如返回按钮样式、标题背景、标题颜色等），应尽可能地提供丰富的属性，这和 Android、iOS 的自定义控件属性的原理是一样的。例如，在本例中，提供的属性如下：

// 导航条属性

```
static propTypes = {
  style: View.propTypes.style,
  titleLayoutStyle: View.propTypes.style,
  navigator: PropTypes.object,
  leftButtonTitle: PropTypes.string,
  popEnabled: PropTypes.bool,
  onLeftButtonClick: PropTypes.func,
  titleColor: PropTypes.string || '#ffffff',
  title: PropTypes.string,
  titleView: PropTypes.element,
  hide: PropTypes.bool,
  statusBar: PropTypes.shape(StatusBarShape),
  rightButton: PropTypes.oneOfType([
    PropTypes.shape(ButtonShape),
    PropTypes.element,
  ]),
  leftButton: PropTypes.oneOfType([
    PropTypes.shape(ButtonShape),
    PropTypes.element,
  ]),
}
```

接下来，只需要按照常见的导航栏样式在 render() 绘制即可（注意，需要提高状态栏的相关属性）。NavigationBar 的完整代码如下：

```
import React, {Component, PropTypes} from 'react';

import {
  StyleSheet, Platform, TouchableOpacity, StatusBar, Text, View
} from 'react-native'

const NAV_BAR_HEIGHT_IOS = 40;
const NAV_BAR_HEIGHT_ANDROID = 50;
const STATUS_BAR_HEIGHT = 20;
```



```

const ButtonShape = {
  title: PropTypes.string.isRequired,
  style: PropTypes.any,
  handler: PropTypes.func,
};

const StatusBarShape = {
  barStyle: PropTypes.oneOf(['light-content', 'default', '']),
  networkActivityIndicatorVisible: PropTypes.bool,
  showHideTransition: PropTypes.oneOf(['fade', 'slide']),
  hidden: PropTypes.bool,
  translucent: PropTypes.bool,
  backgroundColor: PropTypes.string,
  animated: PropTypes.bool
};

export default class NavigationBar extends Component {
  // 构造函数及默认的状态
  constructor(props) {
    super(props);
    this.state = {
      title: '',
      popEnabled: true,
      hide: false
    };
  }

  static propTypes = {
    style: View.propTypes.style,
    titleLayoutStyle: View.propTypes.style,
    navigator: PropTypes.object,
    leftButtonTitle: PropTypes.string,
    popEnabled: PropTypes.bool,
    onLeftButtonClick: PropTypes.func,
    titleColor: PropTypes.string || '#ffffff',
    title: PropTypes.string,
    titleView: PropTypes.element,
    hide: PropTypes.bool,
    statusBar: PropTypes.shape(StatusBarShape),
    rightButton: PropTypes.oneOfType([
      PropTypes.shape(ButtonShape),
      PropTypes.element,
    ]),
    leftButton: PropTypes.oneOfType([
      PropTypes.shape(ButtonShape),

```

```

        PropTypes.element,
      ]),
    },
  },
  static defaultProps = {
    statusBar: {
      barStyle: 'default',
      hidden: false,
      translucent: false,
      animated: false,
    },
  },
}

// 按钮元素属性 (左边和右边的按钮)
getButtonElement(data = {}, style) {
  return (
    <View style={styles.navBarButton}>
      {(!!data.props) ? data : (
        <NavBarButton
          title={data.title}
          style={[data.style, style,]}
          tint_color={data.tintColor}
          handler={data.handler}/>
      )} </View>
  );
}

// 绘制界面
render() {
  let statusBar = !this.props.statusBar.hidden ?
    <View style={styles.statusBar}>
      <StatusBar {...this.props.statusBar} barStyle="light-content"
style={styles.statusBar}/>
    </View> : null;

  let titleView = this.props.titleView ? this.props.titleView :
    <Text style={[styles.title, {color: this.props.titleColor}]}
    ellipsizeMode="head"
    numberOfLines={1}>{this.props.title}</Text>;

  let content = this.props.hide ? null :
    <View style={styles.navBar}>
      {this.getButtonElement(this.props.leftButton)}
      <View style={[styles.navBarTitleContainer, this.props.
titleLayoutStyle]}>
        {titleView}
      </View>
    </View>

```

```

        {this.getButtonElement(this.props.rightButton, {marginRight:
8,})}}
        </View>;
    return (
        <View style={[styles.container, this.props.style]}>
            {statusBar}
            {content}
        </View>
    )
}
}

class NavBarButton extends Component {
    render() {
        const {style, tintColors, margin, title, handler} = this.props;

        return (
            <TouchableOpacity style={styles.navBarButton} onPress={handler}>
                <View style={style}>
                    <Text style={[styles.title, {color: tintColors,,}]> {title} </
Text>
                    </View>
                </TouchableOpacity>
            </View>
        );
    }
}

// 样式表
const styles = StyleSheet.create({
    container: {
        backgroundColor: '#06C1AE',
    },
    navBar: {
        flexDirection: 'row',
        alignItems: 'center',
        justifyContent: 'space-between',
        height: Platform.OS === 'ios' ? NAV_BAR_HEIGHT_IOS : NAV_BAR_
HEIGHT_ANDROID,
    },
    navBarTitleContainer: {
        alignItems: 'center',
        justifyContent: 'center',
        position: 'absolute',
        left: 40,
        top: 0,

```

```

        right: 40,
        bottom: 0,
    },
    title: {
        fontSize: 18,
    },
    navBarButton: {
        alignItems: 'center',
    },
    statusBar: {
        height: Platform.OS === 'ios' ? STATUS_BAR_HEIGHT : 0,
    },
});

```

在其他界面要使用上面封装的 `NavigationBar` 导航条，首先需要导入组件，然后设置一些必要的属性即可。例如，下面的代码显示一个绿底白字的导航条：

```

import NavigationBar from '../component/NavigationBar'
// 在代码中使用
<NavigationBar
  navigator={this.props.navigator}
  popEnabled={false}
  leftButton={ViewUtils.getLeftButton(()=>this.onBackPressed())}
  title=' 城市选择 '
  titleColor='#ffffff'
/>

```

在软件开发中，将具有相同或者相似功能的视图封装为自定义组件，在具体实施过程中，应尽可能多地考虑使用场景，然后通过内置更多的属性来切换场景。

11.3.4 WebView封装

随着移动互联网浪潮的兴起，各种 APP 层出不穷，快速的业务扩展要求团队提高开发效率。在这种情况下，单纯使用纯原生开发似乎成本有点过高，而 H5 的低成本、高效率、跨平台等特性马上被利用起来，从而形成了一种新的开发模式——Hybrid APP。

作为一种混合开发模式，Hybrid APP 底层依赖于 Native 提供的容器，上层使用 HTML、CSS 和 JavaScript 做业务开发。底层透明化、上层多样化，这种场景非常有利于前端工程师介入，也非常适合业务快速迭代。当前，仍然有很多 APP 在使用这种开发模式。

在实际项目开发中，为了更好地使用 `WebView`，一般会对其进行二次封装，以达到方便使用的目的。在封装过程中，通常会将头部导航条和 `WebView` 加载的 URL 地址提供给用户使用。通过传入网页的 URL 地址和导航条信息 `WebView` 就可以自动加载网页内容。

使用封装好的 `WebView` 加载 H5 活动页面的效果如图 11-6 所示。



图11-6 使用WebView加载活动页面

封装后的 WebViewPage.js 相关代码如下：

```
//WebView 封装
import React, {Component} from 'react'
import {
  StyleSheet,WebView,View,
} from 'react-native'

import NavigationBar from '../component/NavigationBar'
import ViewUtils from '../util/ViewUtils'

const WEBVIEW_REF = 'webview';

export default class WebViewPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      url: this.props.url,
      canGoBack: false,
      title: this.props.title,
      theme: this.props.theme
    }
  }

  onBackPress(e) {
```

```

    if (this.state.canGoBack) {
      this.refs[WEBVIEW_REF].goBack();
    } else {
      this.props.navigator.pop();
    }
  }

  onNavigationStateChange(navState) {
    this.setState({
      canGoBack: navState.canGoBack,
      url: navState.url,
    });
  }

  render() {
    return (
      <View style={styles.listViewContainer}>
        <NavigationBar
          navigator={this.props.navigator}
          style={{backgroundColor: '#ffffff'}}
          popEnabled={false}
          leftButton={ViewUtils.getLeftGreenButton(() => this.
onBackPressed())}

          rightButton={ViewUtils.getShareGreenButton()}
          titleColor='#000000'
          title={this.state.title}
        />

        <WebView
          ref={WEBVIEW_REF}
          startInLoadingState={true}
          onNavigationStateChange={(e)=>this.onNavigationState
Change(e)}

          source={{uri: this.state.url}}/>
        </View>
      );
    );
  }

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      backgroundColor: '#ffffff',
    },
    listViewContainer: {
      flex: 1,
      backgroundColor: '#f3f3f4',
    },
  });

```

```

    }
  });

```

在 `WebViewPage` 构造函数中，主要提供 `url`、`canGoBack`、`title` 和 `theme` 4 个状态的监听。在实际使用 `WebViewPage` 的时候，需要给 `WebViewPage` 传递 `url`、`title` 等值。需要注意的是，`WebView` 默认的加载进度条是圆形的，如果读者希望显示加载的进度，可以使用横向进度条。调用 `WebView` 组件的代码如下：

```

// 导包
import WebViewPage from '../component/WebViewPage'
// 调用 WebViewPage 组件显示活动
// 省略...
this.props.navigator.push({
  component: WebViewPage,
  args: {
    url: url,
    title: title,
    ...this.props
  }
})

```

在具体使用 `WebViewPage` 的时候，使用属性传递方式将其值传递给 `WebViewPage`，那么 `WebViewPage` 就可以获取传递过来的对象属性了。

11.3.5 字体样式工具类

在大型移动应用开发中，为了统一管理应用中涉及的字体样式、背景、图片资源等，一般会将这些属性单独写到工具类文件中。在原生开发中，系统为开发者提供了相应的配置文件，而在 `React Native` 中，需要开发人员编写相关的代码进行管理。下面以字体样式为例：

```

import React from 'react';
import { StyleSheet, Text, ReactElement } from 'react-native';

export function HeadingBig({style, ...props}: Object): ReactElement {
  return <Text style={[styles.h40, style]} {...props} />
}

export function Text16({style, ...props}: Object): ReactElement {
  return <Text style={[styles.h16, style]} {...props} />
}

const styles = StyleSheet.create({
  h40: {
    fontSize: 40,
    color: '#06C1AE',
  },

```

```
h16: {  
    fontSize: 16,  
    fontWeight: 'bold',  
    color: '#222222',  
},  
});
```

然后在其他组件中，先导入组件，然后使用相应的标签包裹即可。使用示例代码如下：

```
import {Text16, HeadingBig} from '../component/Text'  
// 省略...  
<Text16 style={{color: '#06C1AE'}}>你好</Text16>
```

11.4 功能开发

当程序的主要技术选型和项目已经搭建完成之后，接下来就进入了模块功能开发阶段。在一些大型的系统开发过程中，往往需要将系统按照模块拆分为多个子模块，然后分给相应的开发人员开发维护。使用模块化开发，可以在一定程度上解决程序的耦合问题，提高开发效率。

11.4.1 分类导航入口开发

在移动应用设计中，首页的设计尤为重要，不仅需要展示尽可能多的商品信息，还应做到繁而不杂。所以，在一些平台级应用中，首页设计过程中一般都会涉及导航分类，如图 11-7 所示。



图 11-7 分类导航入口页面

如图 11-7 所示，分类导航页面一般会根据后台的配置显示相应的模块，并以九宫格的方式展示，当超过一屏显示数量时，会显示指示点。那么，要实现上面的显示效果，需要怎么做呢？

思路分析

在九宫格界面的实现上，有很多方式可以采用，比如使用 ListView 控件或者自定义控件来实现。不管选择哪种方式，其原理都是一样的：通过数据循环绘制子视图。在本例中，我们选择自定义控件来实现九宫格效果。在图例中，每页显示 10 个，每行显示 5 个，而每一行的单个 Item 视图主要由图片和文字构成。基于对页面的分析，子视图 ItemView 的代码如下：

```

import React, { Component } from 'react';
import { StyleSheet, Dimensions, Image, Text, TouchableOpacity } from 'react-native';

const { width } = Dimensions.get('window');

export default class MenuItem extends Component {
  render() {
    return (
      <TouchableOpacity style={styles.container}
        onPress={this.props.onPress}>
        <Image source={this.props.icon} resizeMode='contain' style=
{styles.icon} />
        <Text style={styles.text}>{this.props.title}</Text>
      </TouchableOpacity>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    justifyContent: 'center',
    alignItems: 'center',
    width: width / 5,
    height: width / 5,
  },
  icon: {
    width: width / 9,
    height: width / 9,
    margin: 5,
  },
  text: {
    fontSize: 12,
    color: '#666666',
    justifyContent: 'center',
  }
});

```

然后, 根据接口的数据使用循环绘制的方式绘制界面即可。相关逻辑代码如下:

```

import MenuItem from './MenuItem'
// 省略...
let menuItems = []
// 数据源
let menuInfos = this.props.menuInfos
for (let i = 0; i < menuInfos.length; i++) {
  let menuInfo = menuInfos[i]
  let menuItem = (

```

```

<MenuItem
  key={menuInfo.title}
  title={menuInfo.title}
  icon={menuInfo.icon}
  onPress={() => {
    if (this.props.onMenuSelected) {
      this.props.onMenuSelected(i)
      // this.props.onMenuSelected(menuInfo.title)
    }
  }} />
)
menuItems.push(menuItem)
}

```

其中，数据源实体类格式如下（包含标题和 icon 信息）：

```

menuInfo: [
  { title: '美食', icon: require('../images/icon_homepage_foodCategory.png') },
  ...
],

```

关于分页功能的详细实现，可以参考随书源码，本处不再详述。为了方便代码的维护，可以将上述代码封装成模块，然后在使用的時候直接调用即可。

11.4.2 专题活动开发

活动运营一直都是运营人员提高销售量的最重要的营销手段之一。为了增加用户活跃度，增强产品粘性，提升转化率，团购平台会联合商家不时地推出专题活动吸引用户消费，如图 11-8 所示。



图11-8 专题运营活动

按照上一节实现导航模块的思路，可以将专题活动看成是一个九宫格，具体实现上，只需要将九宫格的 Item 按照设计要求绘制界面即可。需要注意的是，在 iOS 中，涉及 HTTP 请求的地方需要换成 HTTPS。

在开发中，当我们将专题活动的 GridView 视图封装完后，接下来需要使用 fetch 发送网络请求，根据后台返回的数据格式赋值给具体的视图对象。接口返回数据格式如图 11-9 所示。

```

{
  stid: "036080739684176109157139058381894501228",
  - data: [
    - {
      typeface_color: "#ff9900",
      position: 0,
      module: false,
      maintitle: "品质游乐",
      type: 1,
      deputytitle: "领200元券",
      solds: 0,
      id: 21100,
      + share: {...},
      title: "品质游乐",
      deputy_typeface_color: "#21c0ae",
      tplurl: "imeituan://www.meituan.com/web?url=https%
      imageurl: "http://pl.meituan.net/w.h/feop/f78f0966
    },
    + {...},
    + {...},
    + {...}
  ],
  - server: {
    time: 1502086236
  },
  - paging: {
    count: 4
  }
}

```

图11-9 返回数据格式

为了方便使用, <ActiveView> 自定义组件提供了一个 constructor 方法, 用于接收返回的活动数据的 data 数组, 通过自定义组件 <ActiveView> 的构造函数将值传递过去即可, 相关代码如下:

```
export default class HomeScreen extends Component {
```

```

  state: {
    actives: Array<Object>,
  };

```

```

  constructor(props: Object) {
    super(props)
    this.state = {
      actives: [],
    }
  }

```

```
// 使用 fetch 获取活动数据
```

```

requestActives() {
  fetch(api.actives)
    .then((response) => response.json())
    .then((json) => {
      console.log(JSON.stringify(json));
      this.setState({actives: json.data });
    })
    .catch((error) => {

```

```

        console.log('fetch error:'+error);
    })
}

// 使用 ActiveView 控件绘制活动界面
render() {
    return (
        // 省略...
        <ActiveView infos={this.state. actives } onGridSelected={({index) => this.
onGridSelected(index)} />
        // 省略...
    );
}
}

```

在活动模块，当点击某个 Item 之后，就会跳转到 WebView 活动详情页，使用 WebView 加载 H5 活动是 Hybrid APP 开发的常见模式。

11.4.3 商品列表开发

在移动应用开发中，列表是一种很常见的视图表现方式。在实际应用中，通过给列表新增上拉加载更多、新增条件筛选的方式，还可以让用户浏览尽可能多的有用信息，并最终找到自己满意的产品或服务。

开发列表的第一步就是根据设计效果封装列表项。列表项是列表开发中最基本的元素，列表视图就是通过循环地绘制列表项来完成页面的绘制。在本例中，列表项主要由商品的图片、标题、商家介绍、价格（活动价格）组成，如图 11-10 所示。



七八冷面

[4店通用]正宗朝鲜现压冷面1份，邀您共享

22元

图11-10 商品列表项界面

列表长片 ProductItemCell.js 的相关代码如下：

```

import React, { Component } from 'react';
import { View, StyleSheet, TouchableOpacity, Image } from 'react-native';
import { Text16, Paragraph } from '../component/Text'
export default class ProductItemCell extends Component {

    render() {
        let { info } = this.props;
        let imageUrl = info.imageUrl.replace('w.h', '160.0').replace ('http',
'https');
    }
}

```



```

    return (
      <TouchableOpacity style={styles.container} onPress={() => this.
        props.onPress()}>
        <Image source={{ uri: imageUrl }} style={styles.icon} />

        <View style={styles.rightContainer}>
          <Text16>{info.title}</Text16>
          <View>
            </View>
            <Paragraph numberOfLines={0} style={{ marginTop: 8 }}>
{info.subtitle}</Paragraph>
            <View style={{ flex: 1, justifyContent: 'flex-end' }}>
              <Text16 style={styles.price}>{info.price} 元</Text16>
            </View>
          </View>
        </TouchableOpacity>
      );
    }
  }

  const styles = StyleSheet.create({
    container: {
      flex: 1,
      flexDirection: 'row',
      padding: 10,
      borderBottomWidth: 1,
      borderColor: '#e9e9e9',
      backgroundColor: 'white',
    },
    icon: {
      width: 80,
      height: 80,
      borderRadius: 5,
    },
    rightContainer: {
      flex: 1,
      paddingLeft: 20,
      paddingRight: 10,
    },
    price: {
      color: '#06C1AE'
    }
  });

```

然后，使用封装的 RefreshListView 下拉刷新组件填充界面接口，填充的数据使用 fetch

从网络获取。RefreshListView 组件基于 ListView 实现，所以使用时需要注意 dataSource 和 renderRow 函数。使用 RefreshListView 填充界面的代码如下：

```
// 列表界面填充
state: {
  dataSource: ListView.DataSource
};

constructor(props: Object) {
  super(props)
  let ds = new ListView.DataSource({ rowHasChanged: (r1, r2) => r1 !== r2 })
  this.state = {
    dataSource: ds.cloneWithRows([]),
  }
}

requestData() {
  this.requestRecommend()
}

requestRecommend() {
  // 使用 fetch 请求数据
  ...
}

<RefreshListView
  ref='listView'
  dataSource={this.state.dataSource}
  renderRow={(rowData) =>
    <ProductItemCell
      info={rowData}
      onPress={() => {
        this.toDetail()
      }}
    />
  }
  // 下拉重新请求数据
  onHeaderRefresh={() => this.requestData()} />
```

11.4.4 详情页面开发

详情页面是用户了解商品的最终页面，用户在这个页面可以获取商品的主要信息。一个典型的团购商品详情页面如图 11-11 所示。



图11-11 团购详情页

团购详情 DetailPage.js 代码如下:

```
import React, {Component} from 'react';
import {
  Text,StyleSheet,Dimensions,Image,Animated,ScrollView,View
} from 'react-native';
import SpacingView from '../component/SpacingView'
import {Text16, Text14, Paragraph, HeadingBig} from '../component/Text'
import Separator from '../component/Separator'
import NavigationBar from '../component/NavigationBar'
import ViewUtils from '../util/ViewUtils'

const {width} = Dimensions.get('window');
import px2dp from '../util/Utils'

export default class DetailPage extends Component {

  constructor(props) {
    super(props);
    this.state = {
      canGoBack: false,
    }
  }
}
```

```

onBackPress(e) {
  this.props.navigator.pop();
}

onSharePress(e) {
  alert(' 分享 ')
}

renderHeaderView() {
  return (
    <View>
      <View style={styles.topContainer}>
        <Text16 style={{color: '#06C1AE'}}> ¥ </Text16>
        <HeadingBig style={{marginBottom: -8}}>38.5</
HeadingBig>
        <Paragraph style={{marginLeft: 10}}> 门市价: ¥ {(38.5
* 1.1).toFixed(0)}</Paragraph>
        <View style={{flex: 1}}/>
        <View style={styles.styleSubmit}>
          <Text style={styles.submit}>
            立即抢购
          </Text>
        </View>
      </View>
      <Separator />
      <View style={styles.tagContainer}>
        <Image style={{width: 20, height: 20}}
          source={require('../images/icon_deal_anytime_
refund.png')} />
        <Paragraph style={{color: '#89B24F'}}> 随时退 </
Paragraph>
        <View style={{flex: 1}}/>
        <Paragraph> 已售 {1234}</Paragraph>
      </View>
    </View>
  )
}

renderScoreView() {
  return (
    <View style={styles.scores}>
      <Image source={require('../images/ic_star2.png')} style=
{{height: 12, width: 60}}/>
      <View style={{flex: 1}}/>
      <Paragraph style={{color: '#999999'}}> 暂无评分 </Paragraph>
    </View>
  )
}

```



```

        </View>
    )
}

renderBusinessInfoView() {
    return (
        <View style={styles.businessInfo}>
            <Paragraph style={{paddingBottom: 15}}> 商家信息 </Paragraph>
            <Separator />
            <Text14 style={{color: '#000000', paddingTop: 15, paddingBottom:
10}}> 真功夫 (人民广场旗舰店) </Text14>
            <Paragraph> 金陵东路 xx 号 </Paragraph>
        </View>
    )
}

renderNoticeView() {
    return (
        <View style={styles.businessInfo}>
            <Paragraph style={{paddingBottom: 15}}> 购需须知 </Paragraph>
            <Separator />
            <Paragraph style={{color: '#ff6000', paddingTop: 15,
paddingBottom: 10}}> 有效期: </Paragraph>
            <Text14 style={{color: '#000000'}}> 2017.6.15 至 2017.10.15 </
Text14>
            <Paragraph style={{color: '#ff6000', paddingTop: 15,
paddingBottom: 10}}> 使用时间: </Paragraph>
            <Text14 style={{color: '#000000'}}> 10: 00-22: 00 </Text14>
            <Paragraph style={{color: '#ff6000', paddingTop: 15,
paddingBottom: 10}}> 使用规则: </Paragraph>
            <Text14 style={{color: '#000000'}}> . 请提前两天预定 </Text14>
            <Text14> . 每天最多使用 10 张优惠券 </Text14>
            <Text14> . 提供 WIFI </Text14>
        </View>
    )
}

render() {
    return (
        <View style={styles.container}>
            <ScrollView>
                <Animated.View
                    style={{flexDirection: "row", opacity: this.state.
headOpacity}}>
                    <Image source={require('../images/ic_detail.png')}
style={styles.imageStyle}/>

```

```

</Animated.View>

<NavigationBar
  navigator={this.props.navigator}
  popEnabled={false}
  style={{backgroundColor: "transparent", position: "absolute", top:
0, width}}

    leftButton={ViewUtils.getLeftButton(() => this.props.
navigator.pop())}

    rightButton={ViewUtils.getShareButton(() => this.
onSharePress())}/>

  {this.renderHeaderView()}
  <SpacingView />
  {this.renderScoreView()}
  <SpacingView />
  {this.renderBusinessInfoView()}
  <SpacingView />
  {this.renderNoticeView()}
  <SpacingView />

</ScrollView>
</View>
);
}
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'white',
  },
  banner: {
    width: width,
    height: width * 0.5
  },
  topContainer: {
    padding: 10,
    flexDirection: 'row',
    alignItems: 'flex-end',
  },
  imageStyle: {
    width: width,
    height: px2dp(220),
    resizeMode: "cover"
  }
});

```

```

    },
    buyButton: {
        backgroundColor: '#fc9e28',
        width: 94,
        height: 36,
        borderRadius: 7,
    },
    tagContainer: {
        flexDirection: 'row',
        padding: 10,
        alignItems: 'center'
    },
    scores: {
        flexDirection: 'row',
        alignItems: 'center',
        height: 38,
        paddingLeft: 15,
        paddingRight: 15
    },
    businessInfo: {
        padding: 15,
    },
    styleSubmit: {
        marginLeft: 10,
        marginRight: 10,
        backgroundColor: '#fc9e28',
        width: 94,
        height: 36,
        borderRadius: 5,
        justifyContent: 'center',
        alignItems: 'center',
    },
    submit: {
        fontSize: 18,
        color: '#ffffff',
    },
});

```

对于数据来源方面，详情页面根据上一页传递过来的参数请求后端数据，然后填充界面。此页面以信息展示为主，并不会涉及太多的难点。

11.4.5 Modal分享弹窗开发

在很多移动产品中，为了完成社交化属性，一般都会集成一些第三方开放平台，如微信、

微博。开发上通过接入第三方 SDK 完成社交化分享，本文不做这方面的讲解，本文要介绍的是使用 React Native 的 Modal 组件完成分享界面的开发，分享界面如图 11-12 所示。

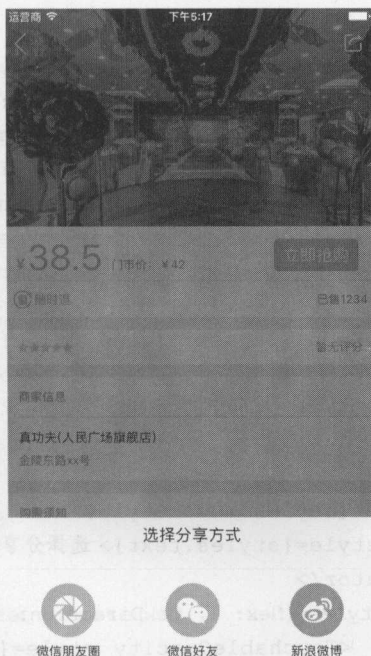


图11-12 社交化分享

Modal 组件提供了 visible 属性来控制模态是否可见。当点击窗口外的区域，弹窗会自动消失。整个分享弹窗可以看做一个界面，这个界面包含两部分：上半部分的透明部分和下半部分的分享部分。为了实现点击界面上半部分或点击某个分享按钮界面消失的效果，还需要复写 componentWillReceiveProps() 方法。具体代码如下：

```
// 自定义分享弹窗
import React, {Component} from 'react';
import {View, TouchableOpacity, Alert, StyleSheet, Dimensions, Modal, Text,
Image} from 'react-native';
import Separator from "../Separator";

const {width, height} = Dimensions.get('window');
const dialogH = 110;

export default class ShareAlertDialog extends Component {

  constructor(props) {
    super(props);
  }
}
```



```

    this.state = {
      isVisible: this.props.show,
    };
  }

  componentWillReceiveProps(nextProps) {
    this.setState({isVisible: nextProps.show});
  }

  closeModal() {
    this.setState({
      isVisible: false
    });
    this.props.closeModal(false);
  }

  renderDialog() {
    return (
      <View style={styles.modalStyle}>
        <Text style={styles.text}> 选择分享方式 </Text>
        <Separator/>
        <View style={{flex: 1, flexDirection: 'row', marginTop: 15}}>
          <TouchableOpacity style={styles.item} onPress={() =>
            Alert.alert(' 分享到微信朋友圈 ')}>
            <Image resizeMode='contain' style={styles.image}
              source={require('../images/share_ic_friends.
png')}}/>
            <Text> 微信朋友圈 </Text>
          </TouchableOpacity>
          <TouchableOpacity style={styles.item}>
            <Image resizeMode='contain' style={styles.image}
              source={require('../images/share_ic_weixin.
png')}}/>
            <Text> 微信好友 </Text>
          </TouchableOpacity>
          <TouchableOpacity style={styles.item}>
            <Image resizeMode='contain' style={styles.image}
              source={require('../images/share_ic_weibo.
png')}}/>
            <Text> 新浪微博 </Text>
          </TouchableOpacity>
        </View>
      </View>
    )
  }

```

```

}

render() {

  return (
    <View style={{flex: 1}}>
      <Modal
        transparent={true}
        visible={this.state.isVisible}
        animationType='fade'
        onRequestClose={() => this.closeModal()}>
        <TouchableOpacity style={styles.container} activeOpacity={1}
          onPress={() => this.closeModal()}>
          {this.renderDialog()}
        </TouchableOpacity>
      </Modal>
    </View>
  );
}

```

```

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: 'rgba(0, 0, 0, 0.5)',
  },
  modalStyle: {
    position: "absolute",
    top: height - 170,
    left: 0,
    width: width,
    height: dialogH,
    backgroundColor: '#ffffff'
  },
  subView: {
    width: width,
    height: dialogH,
    backgroundColor: '#ffffff'
  },
  text: {
    flex: 1,
    fontSize: 18,
    margin: 10,
    justifyContent: 'center',

```

```

      alignItems: 'center',
      alignSelf: 'center'
    },
    item: {
      width: width / 3,
      height: 100,
      alignItems: 'center',
      backgroundColor: '#ffffff'
    },
    image: {
      width: 60,
      height: 60,
      marginBottom: 8
    },
  });

```

在其他需要分享的界面中，先导入组件，当点击分享按钮弹出分享窗口即可。当点击某个分享平台，分享到相应的平台，分享也可以使用第三方库提供的功能。需要注意的是，在封装弹窗类组件的时候，由于手机的分辨率不一样，还会涉及分辨率的适配问题。

11.5 完成开发

功能的开发完成并不意味着移动应用开发的完结，当功能模块开发完成之后，还需要针对平台做一些简单的配置以及打包等操作，最后提交应用市场供用户下载使用。

11.5.1 添加闪屏页

在很多应用冷启动的过程中，每次都会给用户一个过渡页面。这个过渡页面一方面可以缓解用户等待的焦虑情绪，另一方面通过在闪屏页加载广告也可以达到营销的目的。例如，经典的微信闪屏页就是一个人在看地球。

典型的闪屏页，往往包含广告展示、公司 logo、闪屏倒计时等元素，合理地设置闪屏页，不仅可以提升用户体验，还可以加强应用的宣传。示例代码如下：

```

import React, {Component} from 'react';
import {
  StyleSheet, View, Image, Dimensions, ActivityIndicator, StatusBar
} from 'react-native';
import MainScreen from '../main/MainScreen';
let {width, height} = Dimensions.get("window");

export default class SplashView extends Component {

```

```

constructor(props) {
  super(props);
  this.state = {
    animating: true, // 默认显示加载动画
  };

  // 倒计时 3 秒后进入首页
  componentDidMount() {
    setTimeout(() => {
      this.props.navigator.replace({
        component: MainScreen,
      });
    }, 3000);
  }

  render() {
    return (
      <View style={styles.container}>
        <StatusBar hidden={true}/>
        <Image style={styles.splash} source={require('../images/splash.png')} resizeMode={'cover'} />
        <ActivityIndicator
          animating={this.state.animating}
          style={[styles.centering, {height: 70}]}
          size="small" />
      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
  },
  splash: {
    width: width,
    height: height,
  },
  centering: {
    flex: 1,
    marginTop: -height,
    alignItems: 'center',
    justifyContent: 'center',
  },
});

```



```
padding: 8,
},
});
```

上述代码中，通过调用 `setTimeout` 定时器进行倒计时，倒计时完成后即跳转到指定的页面。使用 `ActivityIndicator` 指示器可实现进度加载效果。需要注意的是，添加闪屏页之后，需要修改默认启动页。

11.5.2 修改应用图标和名称

修改应用图标和名称是产品上线前的最后工作，应用图标代表着公司形象，一个好的图标能向用户传达产品的信息，吸引更多的用户使用。应用图标和名称会显示在用户设备的主界面上，也会显示在 App Store 上。如果没有经过修改，那么应用图标和名称就是初始化项目时候的图标和名称，如果需要修改，就需要在项目的原生代码中修改。

Android篇

使用 Android Studio 打开 React Native 项目下的 Android 目录，打开 Android 文件夹下的 `AndroidManifest.xml` 配置文件，找到 `application` 标签，修改 `label` 属性（应用名称）和 `icon` 属性（应用图标）即可。

`AndroidManifest.xml` 文件配置如下：

```
<application
  android:name=".MainApplication"
  android:allowBackup="true"
  android:label="@string/app_name"      // 应用名称
  android:icon="@mipmap/ic_launcher"   // 应用图标
  android:theme="@style/AppTheme">
  // 省略...
</application>
```

当修改完成之后，需要卸载掉之前的 APP，然后重新安装。

iOS篇

使用 Xcode 打开 React Native 项目下的 iOS 目录，打开 `Info.plist` 配置文件，修改 `Bundle name` 属性即可修改应用的名称。工程的 `images.xcassets` 的文件夹是专门用于存储图片的地方。在 React native 开发中，一般将项目的资源放在 React Native 的资源文件目录下，但是对于一些平台配置图片，还是需要放在相关的原生文件目录下。

首先，打开 `images.xcassets` 文件，选择 `AppIcon` 项，然后将准备好的不同分辨率下的图标拖拽到方框内即可。当然，也可以使用一键生成图标工具生成 `AppIcon`。当然，修改应用图标（见图 11-13）、名称等配置信息，需要卸载掉之前的 APP，然后重新安装才能看到效果。

当产品开发完成，达到上线的标准后，就可以打包上传到应用市场以供用户下载了。打包、上线等流程请参照之前章节的讲解，本处不再讲解。

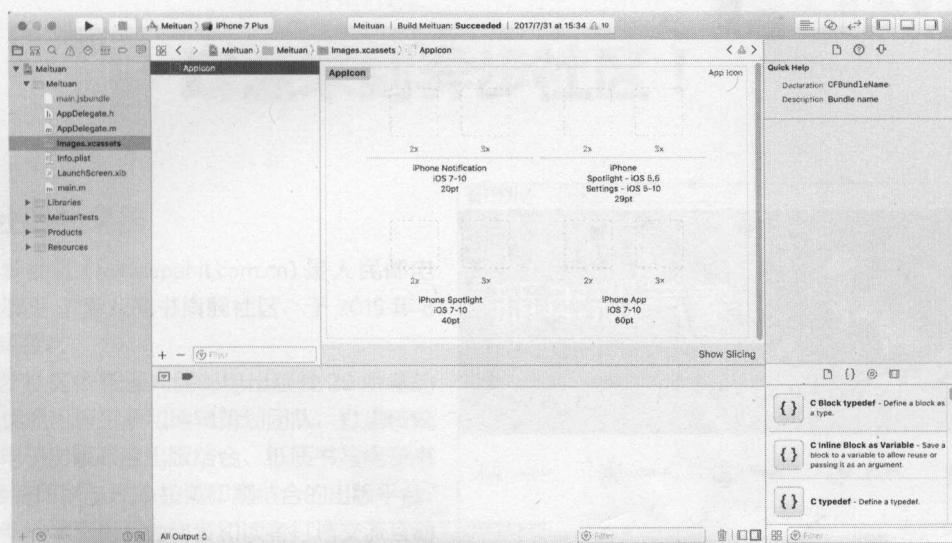


图11-13 更改iOS应用图标

11.6 小结

至此，移动团购应用就开发完成了。由于篇幅所限，本章并没有对项目中的所有技术都做讲解，对于项目上线相关的流程，读者可以参考之前章节的讲解。本项目涉及的源码读者可以参考随书源码学习。

在本项目中，用到了很多第三方库，合理地使用第三方库能节约项目开发时间，提高产品的性能。虽然本项目还达不到商业项目的复杂度，但是作为示例项目，它基本涵盖了移动项目开发的基本流程以及开发的基本思想。学习到此处，是不是已经跃跃欲试了？



React Native

移动开发实战

本书全面详尽地介绍了 React Native 框架的方方面面，内容涵盖：基础知识、环境搭建与调试、开发基础、常用组件、常用 API、组件封装、网络与通信、混合开发、热更新与打包部署，以及两个实际案例的完整开发教程。这些丰富的内容不仅能让读者了解这款框架中涉及的各类概念，还能指导读者的开发实践。

本书适合有一定 Android、iOS 原生开发基础和 CSS 基础的移动开发工程师学习。

向治洪

易居旗下客户端主管，曾供职于携程网、驴妈妈旅游网等互联网公司；

在 Android、iOS 和移动跨平台开发方面有较深的理解和丰富的经验；

云栖社区资深版主，CSND 博客访问量逾 200 万，免费视频教程备受好评。



异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

ISBN 978-7-115-47096-6



ISBN 978-7-115-47096-6

定价：69.00 元

分类建议：网页制作

人民邮电出版社网址：www.ptpress.com.cn